SPECIAL PROJECT REPORT

Data-driven feature extraction and prediction using Deep Neural Networks

Author: Felix Oscar Ærtebjerg (s204797)

Supervisor: Adem Rosenkvist Nielsen Aouichaoui

DTU Chemistry
Department of Chemical and Biochemical Engineering
Technical University of Denmark

June 21, 2024

Contents

1 Introduction

The twentieth century was a revolutionary period of time for medicine and pharmaceutical drugs. Leading into it, chemicals extracted from coal-tar (Jones, 2011) were being made into some of the earliest drugs still available. 1897 then saw the chemist Felix Hoffman inventing the well known drug Aspirin (Jones, 2011; Desborough and Keeling, 2017). The key ingredient of Aspirin stems from the bark of the willow tree, and had already been used for millennia as a pain-reliever, but modern advances allowed the extraction and creation of the still useful Aspirin.

Within a hundred years humans had invented life saving cancer treatments, the MRI machine for early disease detection and much more. These inventions have improved life conditions wherever they have been able to reach, and undoubtedly more is known about medicine than ever.

In recent years, however, pharmaceutical drug discovery has slowed down. There are many factors one could attribute to this, but a very intuitive one is that a lot of the 'general' drugs have been exhausted. Instead, the focus has shifted to finding specific drugs treating distinct ailments. This also means that is has become harder to develop something that is both new and effective. Rather than investigate every compound by hand, an expensive process, scientists have been trying to automate some of the first, labour-intensive steps of drug discovery using Machine Learning, specifically Deep Learning.

Although this is by no means a new idea, modern technology like GPUs (Graphic Processing Units) and TPUs (Tensor Processing Units) have allowed for computation heavy algorithms to be feasible. This hardware, together with the quickly improving field of deep learning, means that soon machine learning will be used for drug discovery and personalized medicine (Vamathevan et al., 2019) in an industrial setting.

In the spirit of using machine learning as a practical tool, we developed a toolbox for streamlined and effective property prediction of graphs, a useful data representation within chemistry and bio-science. Our package, `GraPE-Chem` (**Gra**ph based **P**roperty **E**stimation for **Chem**istry), is freely available on GitHub and as a python package, and a complete documentation together with demonstrations are also available online.

In the following report, some of the foundational Graph Neural Networks that use deep learning tools will first be introduced mathematically and examined. Then, the key features of `GraPE-Chem` are reviewed and justified for the machine learning objective. Some machine learning operations (MlOps) techniques are detailed and why they play an important role in development. The experimental setup used to benchmark the toolbox is given and experiments are conducted on four chemical datasets. The results are evaluated used metrics and various post-analysis tools. The report is then concluded with a discussion of the findings and a perspective over future development.

2 Graph Neural Network Models

In the following section, some of the key GNN models for chemical property prediction are presented in mathematical detail. All of them are also implemented in the `GraPE-Chem` toolbox, and will be evaluated at the end of the report.

2.1 History

Graph Neural Networks (GNNs) are a relatively new part of the machine learning (ML) research space. The seminal Graph Network study by Scarselli et al., 2009 presents a lot of the key features seen in all subsequent papers. For reference, the first work on RNN was by Rumelhart et al., 1985, about 20 years earlier. Although Scarselli et al. laid the foundation, graph neural networks would not see wide spread use until Kipf and Welling, 2017, published their paper on a graph convolutional network. This model was effective enough to generate a new wave of interest for the Graph Neural Networks. Today, GNNs make up a large part of modern ML research, as our data is often already stored in graph-like databases. Industrial problems like transport optimization and drug discovery especially have seen companies invest huge efforts into GNNs. Another, more research orientated endeavor has been the AlphaFold project (Jumper et al., 2021) by Google, pushing the boundaries of bioinformatics.

2.2 MPNN

The first model to be examined is the 'Message Passing Neural Network' by Gilmer et al., 2017. To date, this is one of the most influential papers on graph neural networks with around eight thousand citations according to Google Scholar at the time of writing. This paper cemented a lot of the terminology used in subsequent research publications, and coined the term 'MPNN'-'Message Passing Neural Network', now commonly used to refer to characteristic Graph Neural Network algorithms.

Although multiple models are presented in the paper, the framework presented in the following is the one commonly referred to as MPNN. As seen shortly, there is a lot of freedom in the definition of the model.

To begin, let us clarify what a GNN refers to. Specifically, a Graph Neural Network describes a machine learning algorithm used to embed graph structured data into an arbitrary latent space. Similar to how RNNs or recurrent neural networks encode serial data, the GNN encodes graph data. The specific algorithms needed to capture these types of structures are what set GNNs apart from other neural networks (NNs). With this purpose in mind, the following are the mathematics underlying the MPNN.

2.2.1 Message Passing algorithm

All message passing algorithms consist of three parts: **messages**, **aggregation** and **updates**, all three of which occur sequentially. Put into words, first, messages are generated that are passed along the graph from node to node. These messages are then aggregated at the nodes or edges, which are subsequently updated based on this aggregation and an

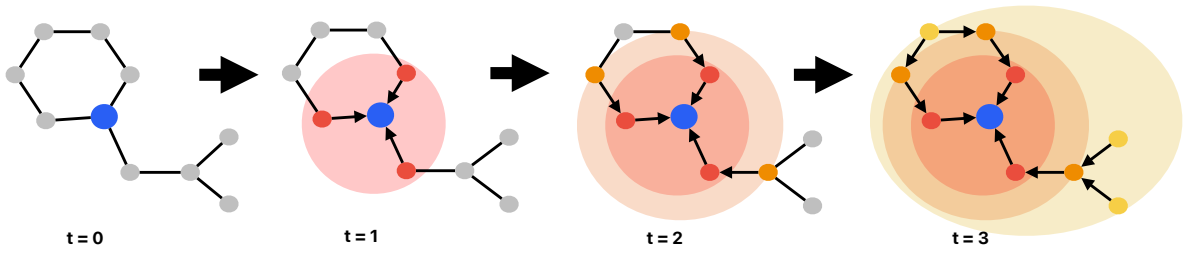


Figure 1: This figure shows an what three steps of the message passing algorithm looks like for one node. The blue node is the target of the aggregation and the red dots are the origins of the messages. The background color indicates what k-hop region we are in. Finally, the arrow indicates, what node representation (origin) is used to update what target. At $t = 2$, for example, the information flows from the orange nodes through the red nodes to be aggregated at the blue node. Also note, that for each step, the messages contain information from the k-hop neighborhood.

update function. Often, the message generating functions are combined with the aggregation directly, usually a sum, to reduce notation clutter.

G is assumed to be a bidirectional graphs [1] with node features x_v for node v and edge features e_{vw} between nodes v and w. Neighbors of node v are defined to be $N(v)$. The message passing algorithm runs for T iterations, and for each step the message generating function is M_t and U_t the node update function. The algorithm at time step t is then:

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \tag{1}$$

$$h_v^{t+1} = U_t(h_v^t, h_w^t, e_{vw}) \tag{2}$$

The initialization and read-out aside, this is the key framework that describes message passing. The main differences between different message passing algorithms is how they construct the message generating and update functions. The one by Gilmer et al., 2017, uses a neural network to generate the message and the gated recurrent unit (GRU; Cho et al., 2014b), to update the hidden representations of the nodes. The idea behind the GRU update is to allow the model to 'forget' features or prevent certain features to be updated, essentially acting as a context filter.

Inserting the MPNN specific terms, the message passing becomes:

[1] A graph being bi-directional is usually a fair assumption, as any graph without information on edge direction can be considered as such.

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) = \sum_{w \in N(v)} A(e_{vw}) h_w \tag{3}$$

$$h_v^{t+1} = U_t(h_v^t, h_w^t, e_{vw}) = \text{GRU}(h_v^t, m_v^{t+1}) \tag{4}$$

where $A()$ is a neural network mapping the edge features of bond e_{vw} to the a $d \times d$ matrix and d denotes the number of hidden features. The weights of $A()$ are shared between the iterations, sometimes referred to as weight tying. The only thing missing now is the initialization and the read-out function. Here, the hidden state h_v^0 is initialized by the node features x_v. It is common to not use the node features directly, but rather to first use a linear layer to project them into the hidden feature dimension d.

And finally, the read-out function is the set2set model by Vinyals et al., 2016. The read-out model serves to combine the hidden states into a singular, graph-level embedding that can be used for classification or regression. To briefly summarize the function of the set2set read-out specifically, at the end of the message passing algorithm the hidden states of each node are extracted, giving the 'context' that represents the overall graph from the nodes. The set2set aggregation function extracts the information using the long short-term memory layer by Hochreiter and Schmidhuber, 1997 for a given number of iteration and then aggregates it into a graph level embedding. This embedding is invariant wrt. the order of nodes and has more expressive power than a regular sum (Gilmer et al., 2017). The invariancy is especially important for the problem at hand: given two graphs with the same nodes and bonds, a model should then produce the same output for both regardless of the node ordering.

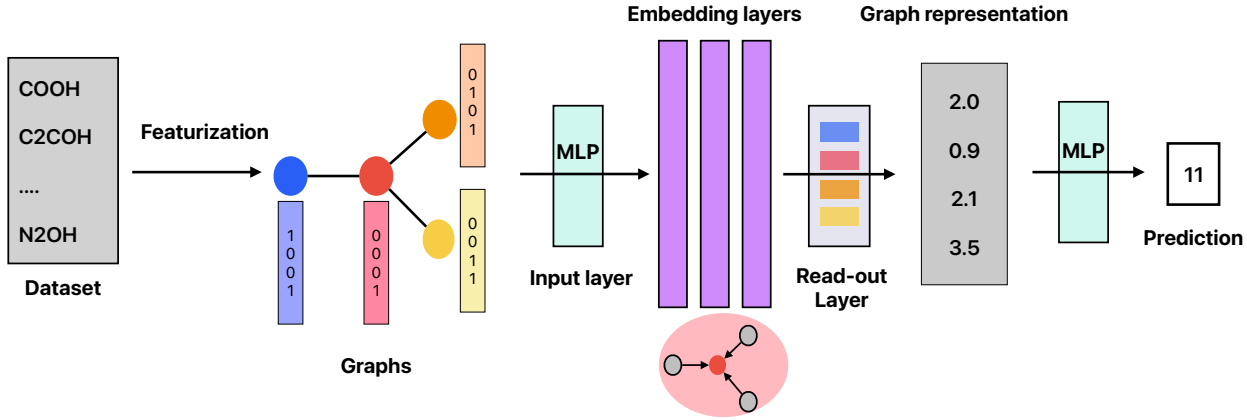


Figure 2: A visualization of what a complete Graph Neural Network model will contain. The steps are as follows: a dataset will first be featurized according to the graph structure and then projected into the hidden representation space using, usually, an MLP. The graphs is then passed through a set of embedding layers, which could be the MPNN, DMPNN or any message passing algorithm. The resulting graph is then aggregated into a graph level representation, containing features describing the singular graph as a whole. Finally, an MLP is used for prediction.

2.2.2 Related studies

The MPNN framework has been instrumental in the development of GNNs and is now often the foundation for more advanced models, often applied to a large variety of subjects. Some interesting applications are for protein sequence design (Dauparas et al., 2022), mapping between function spaces (Kovachki et al., 2023) and equivariant GNNs (Satorras et al., 2021).

2.3 Weave

A model that often is used as a baseline in chemical property estimation is the 'Weave' model by Kearnes et al., 2016 and described by Gilmer et al., 2017 in their own paper. The feature that sets it apart from other message passing algorithms is the use of a hidden edge representation that is updated together with the hidden node representations. This could be especially interesting in the context of generating new molecules, where the edge features determine the bond types.

2.3.1 Message Passing Algorithm

The Weave message passing scheme is mostly made up of linear layers, ReLU activation layers and concatenation of features. Let cat[..., ...] denote the concatenation operation. Like mentioned before, one message passing step is split up in the node and edges part. The hidden node node representation is updated as follows:

$$m_v^{t+1} = M(h_v^t, h_w^t, e_{vw}^t) = \sum_{w \in N(v)}^{N} e_{vw}^t \tag{5}$$

where the generated message is just the edge representations of all bonds of v summed. The update is:

$$h_v^{t+} = U_t(h_v^t, m_v^{t+1}) = \alpha(W_1 \text{ cat}[\alpha(W_0 h_v^t), m_v^{t+1}]) \tag{6}$$

Here, α denotes the ReLU activation function, and W_0, W_1 are learned matrices. The matrix-vector product is then just the definition of a regular 'linear' layer. The hidden edge representation update is:

$$e_{vw}^{t+1} = U_t^{'}(e_{vw}^t, h_v^t, h_w^t) = \alpha(W_4 \, \alpha(\text{cat}\left[\text{cat}[W_2, e_{vw}^t], \alpha(W_3 \text{ cat}[h_v, h_w^t])\right]) \tag{7}$$

where W_2, W_3 and W_4 are again learnable matrices. The edge representation updates is a combination of the current edge features, the source node features and the destination node features. One could also see this as a way of using multilayer perceptron layers (MLPs) to mix the information relevant to the edge information.

2.4 D-MPNN

The, almost natural follow-up to the MPNN is the 'Directed-Message Passing Neural Network' (D-MPNN) by Yang et al., 2019. Their model builds on the MPNN model by Gilmer et al., 2017, and incorporates directed message passing similar to Dai et al., 2016.

The main idea is the following: during message passing in undirected graphs, which are coded as bi-directional rather than non-directional, messages can propagate backwards or into undesired branches. By adding redundant or undesired information from already covered connections and unrelated branches respectively, the undirected message passing potentially introduces noise into the final graph representation (Yang et al., 2019). To avoid this, bonds are represented with directed edges and the message passing algorithm will follow their direction. As mentioned in Yang et al., 2019, and in Dai et al., 2016, this is similar to the propagation in probabilistic graphical models which only propagate into one direction.

To reiterate, the key distinction between D-MPNN and MPNN are the directed edges and that messages are passed on the *edges* rather than just the nodes. So, instead of a hidden node state h_v^t and message m_v^t, the D-MPNN employs hidden states h_{vw}^t and messages m_{vw}^t. Here, v and w denote the origin and destination nodes respectively. This makes the hidden state or message with direction vw distinct from wv.

2.4.1 Message Passing Algorithm

The following passage will examine the exact algorithm of the D-MPNN message passing algorithm following the paper by Yang et al., 2019.

First, the hidden state of the edges is initialized:

$$h_{vw}^0 = \tau(W_i \, \text{cat}(x_v, e_{vw})) \tag{8}$$

where h_{vw}^0 is the initial hidden state, $W_i \in \mathbb{R}^{hxh_i}$ is a learnable weight matrix, x_v are the node features of node v and e_{vw} are the edge features. Lastly, τ is an activation function like ReLU (Nair and Hinton, 2010) or LeakyReLU (Maas et al., 2013) and cat$[x_v, e_{vw}]$ the concatentation operation of the origin node and edge features.

The massages are passed and the hidden representations updated for $t \in \{1, ..., T\}$ steps. The equations are as follows:

$$m_{vw}^{t+1} = \sum_{k \in \{N(v) \backslash w\}} M_t(x_v, x_k, h_{kv}^t) \tag{9}$$

$$h_{vw}^{t+1} = U_t(h_{vw}^t, m_{vw}^{t+1}) \tag{10}$$

Note that $N(v)$ refers to the neighbors of node v, and $\{N(v) \backslash w\}$ are the neighbors without target node w. Here, M_t refers to the message aggregation. This could be any function, but the authors simply let $M_t(x_v, x_w, h_{vw}^t) = h_{vw}^t$. This means that the message m_{vw}^{t+1} is an aggregation of the t-jump edge features. Figure 3 shows the aggregation step visually.

The update function U_t in 10 was chosen to be a simple neural network shared across all steps t:

$$U_t(h_{vw}^t, m_{vw}^{t+1}) = U(h_{vw}^t, m_{vw}^{t+1}) = \tau(h_{vw}^0 + W_m m_{vw}^{t+1}) \tag{11}$$

where $W_m \in \mathbb{R}^{hxh}$ is a learnable weight matrix of hidden feature size h. The addition of h_{vw}^0 to the most recent message m_{vw}^{t+1} is a skip connection (He et al., 2015) to the original edge feature vector of the bond v to w.

After this algorithm has run for T steps, the message passing algorithm is concluded by aggregating the relevant 'final' hidden edge states per node:

$$m_v = \sum_{k \in N(v)} h_{kv}^T \tag{12}$$

and, finally, rebuilding the atom representation:

$$h_v = \tau(W_a \, \text{cat}[x_v, m_v]) \tag{13}$$

as a combination of the original atom features and the 'learned' features. $W_a \in \mathbb{R}^{hxh}$ is again a learnable weight matrix.

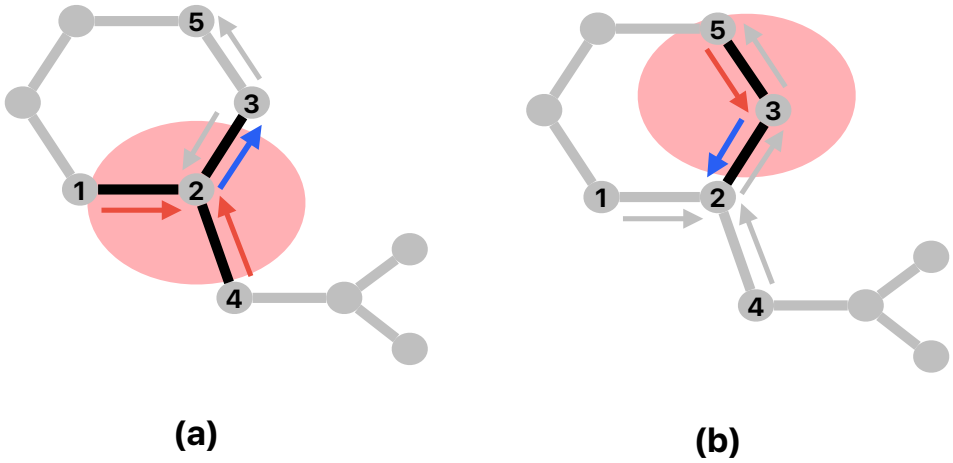


Figure 3: An illustration of the D-MPNN message passing algorithm. Like mentioned in the text, messages are propagated on the directed edges rather than just the nodes. In (a), the blue edge going from 2 to 3 is updated using the edges pointing towards 2 in red. In (b), the edge going from 3 to 2 is updated using the edge from 5 to 3 in red.

2.5 MEGNet

The third model of interest is the 'MatErials Graph Network' (MEGNet) developed by Chen et al., 2019. This is a novel approach to apply machine learning not only to molecule prediction like the MPNN (Gilmer et al., 2017) before it, but also to for crystal prediction. Based on the works of Battaglia et al., 2018, they employ a different message passing algorithm than those discussed before as well as a more complex overall structure.

To stay consistent with previous notation, V is a set of $N(v)$ atoms and h_v as the feature vector of atom v, E as a set of $N(e)$ atom bonds or edges and h_{vw} as the bond from atom v to w.

Additionally, $\mathbf{u}$ is the global state vector, comprising of 'global' features such as the temperature or solubility. Overall, the input graph $(E, V, \mathbf{u})$ is mapped to the output graph $G = (E', V', \mathbf{u}')$.

2.5.1 Message Passing Algorithm

MEGNet's message passing algorithm is structured into three parts, one for each data type (atom, bond or state). In the first part, the atom bonds or edges are updated without prior message aggregation:

$$h'_{vw} = \phi_e(\text{cat}(h_v, h_w, h_{vw}, \mathbf{u})) \tag{14}$$

here, ϕ_e is the bond update function which will be discussed later. Effectively, the bonds are updated with their own features, the connected atom features and the global state vector.

The atoms are updated according to the 'traditional' message passing scheme. First the bond information around the atom i is aggregated:

$$\tilde{h}_i^e = \frac{1}{N_i(e)} \sum_{k=1}^{N_i^e} h_{ik} \tag{15}$$

after which the atom features are updated with the aggregated atom information, the original features and the global state features:

$$h_i' = \phi_v(\text{cat}(\tilde{h}_i^e, x_i, \mathbf{u})) \tag{16}$$

Again, ϕ_v describes the atom update function. Note that x_i are the original features of atom i. After these two updates, the global state features are updated using all atoms and bonds:

$$\tilde{\mathbf{u}}^e = \frac{1}{N(e)} \sum_{k=1}^{N(e)} h_{ik} \tag{17}$$

$$\tilde{\mathbf{u}}^v = \frac{1}{N(v)} \sum_{k=1}^{N(v)} h_k \tag{18}$$

$$\mathbf{u}' = \phi_u(\text{cat}[\tilde{\mathbf{u}}^v, \tilde{\mathbf{u}}^e, \mathbf{u}]) \tag{19}$$

with ϕ_u as the global state update function. The use of the original global feature vector $\mathbf{u}$ acts as a skip connection that helps preserve the global elements of the graph. A visual representation of the algorithm can be found in figure 4.

As for the update functions (ϕ_e, ϕ_v, ϕ_u), Chen et al. used a two layer MLP with softplus activation functions (Zheng et al., 2015). The above is then combined into a MEGNet block that is re-used multiple times throughout the total MEGNet model.

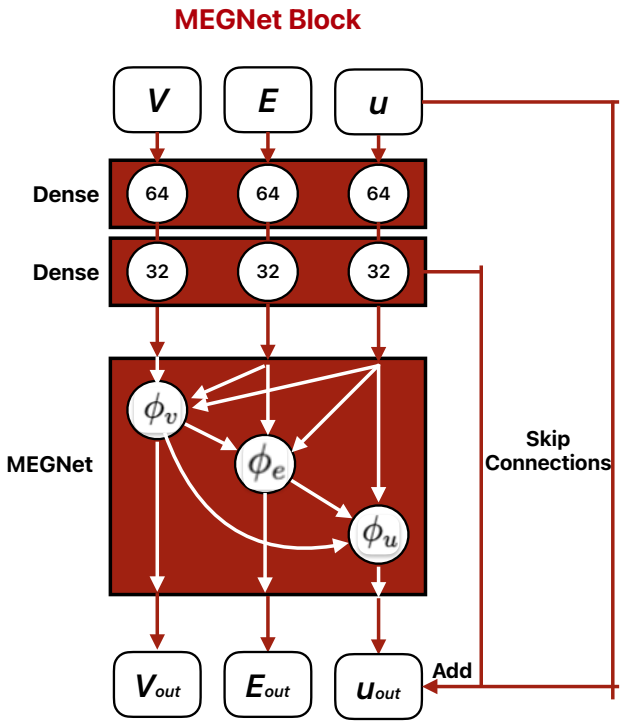


Figure 4: A recreation of the MEGNet block architecture following the description by Chen et al., 2019. Notice that the global state is employed as a skip connection to previous layers, holding the combined node, edge and global state information. The node, edge and state features are first passed through two dense layers in the start of each block, where after messages are generated, aggregated and each set of features is updated sequentially.

2.6 Attentive FP

A big issue with the message passing algorithm used in D-MPNN or MEGNet is the locality associated with the message passing algorithm. The D-MPNN algorithm assigns the molecules very similar importance by ignoring distance for the most part while MEGNet employs an algorithm giving more weight to closer molecules (Xiong et al., 2020). An attempt to find an effective middle ground between the two is the Attentive FP (AFP) network by Xiong et al., 2020. Building on top of the foundation laid by the work of Veličković et al., 2018, and Bahdanau et al., 2016, they employ the graph attention network on molecule prediction. They argue, that graph attention is a very effective way to identify the correct relationship between nodes, regardless of spacial distance. Their model consists of two parts: node-wise attention followed by graph-level attention using a virtual node.

2.6.1 Graph Attention

Like mentioned, the primary algorithm used in the Attentive FP is graph attention (Veličković et al., 2018). This involves the following: For each time step t of T and node v in the aggregation step, there are three sequential operations: *alignment*, *weighting* and *context*. Like in natural language processing, the core idea is to obtain a context for each node that represents the environment around it and its key features.

The algorithm begins with the *alignment* of the nodes:

$$\mathbf{e}_{vw} = \text{leaky-relu}(\mathbf{W} \cdot \text{cat}[\mathbf{h}_v, \mathbf{h}_w]) \tag{20}$$

In this step, the hidden representations of two connecting nodes are compared and the alignment value is extracted, a measure of how close the two nodes are.

Next is the *weighing*:

$$\mathbf{a}_{vw} = \text{softmax}(\mathbf{e}_{vw}) = \frac{\exp(\mathbf{e}_{vw})}{\sum_{\mathbf{u}\in\mathbf{N}(\mathbf{v})} \exp(\mathbf{e}_{vu})} \tag{21}$$

Here, the 'attention' that should be given to each alignment or node-pair is weighted out. The best node-pair *alignment* is given the highest attention or weight for the next step, the *context*:

$$\mathbf{C}_v = \text{elu}\left(\sum_{\mathbf{u}\in\mathbf{N}(\mathbf{v})} \mathbf{a}_{vu} \cdot \mathbf{W} \cdot \mathbf{h}_u\right) \tag{22}$$

The combined *context* of a node is then the aggregation of attention vectors times the weighted neighboring hidden node states. This could be interpreted as a summary of the relevant information that can be extracted from its neighborhood.

2.6.2 Message Passing (1): node-level

Following the original message passing algorithm by Gilmer et al., 2017, the Attentive FP message generation and aggregation step at time step t follows:

$$\mathbf{C}_v^{t-1} = \sum_{\mathbf{u}\in\mathbf{N}(\mathbf{v})} \mathbf{M}^{t-1}(\mathbf{h}_u^{t-1}, h_v^{t-1}) \tag{23}$$

Here, $\mathbf{C_v^{t-1}}$ is the summed context vector of node v, and $\mathbf{M^{t-1}}$ is the graph attention algorithm at time step t between the two hidden representations $\mathbf{h}_v^{t-1}$ and $\mathbf{h}_u^{t-1}$.

As for the update function, Xiong et al. chose the non-linearity GRU or Gated Recurrent Unit (Cho et al., 2014a). It can be thought of as a way to evaluate how much of the aggregated context is relevant to the new hidden state. The actual update is as follows:

$$\mathbf{h}_v^t = \mathrm{GRU}^{t-1}(\mathbf{h}_v^{t-1}, \mathbf{C}_v^{t-1}) \tag{24}$$

where $\mathbf{h}_v^t$ is the new hidden state of node v after t steps. A sketch of the node-level graph attention can be found in (a) of figure 5.

2.6.3 Message Passing (2): graph-level

That was the first block of the attentive FP. So far, the output of each step was an update to all the node hidden states. Previous models like the MPNN or DMPNN would now employ a readout function such as Set2Set or a sum to generate a graph level representation.

The Attentive FP, on the other hand, adds another block of graph attention layers to learn the graph level representation directly. To do so, after T steps of node-level message passing, a hypergraph is formed. Essentially, this is done by introducing a virtual node that is connected to all graph nodes, illustrated in (b) of figure 5, and initialized by averaging all the node hidden states. All other node bonds, ie. the 'real' bonds between atoms, are removed so only the connection to the center remains. This new virtual graph and specifically the virtual node is a representation of the graph as a whole. By performing another set of L graph attention layers like above with the new graph, a context driven featurization of our graph is learned.

The final embedding of the original graph in Attentive FP model is the virtual node after the full graph-level attention block. After passing this through a predictor like an MLP, see (c) in 5, a prediction is generated.

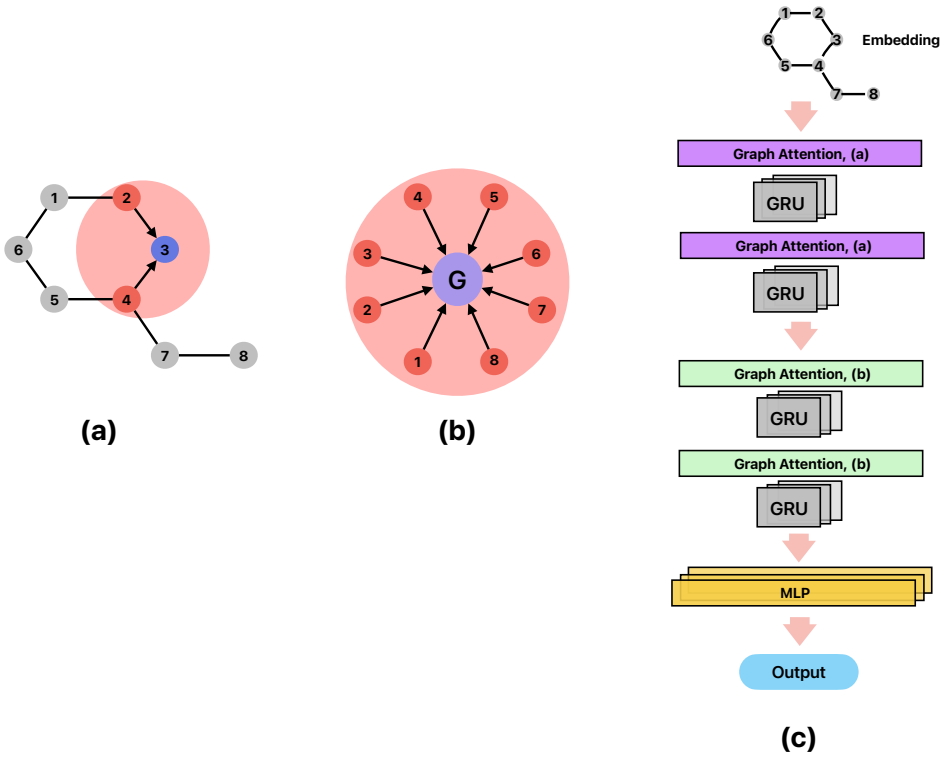


Figure 5: A visualization of the AFP algorithm. The arrows indicate the direction of the information flow. (a) shows the first step, where graph attention is used on the node-level to generate meaningful information at the nodes. (b) shows the second step, here, a virtual node representing the graph as a whole is generated and connected to all nodes. Each message passing phases then uses all the nodes at once, building a useful graph-level representation. And (c) is the model presented as a whole, going through node-level attention, then graph level attention and finally an MLP to produce the output.

2.7 Model Overview

Before the toolbox itself is discussed, table 1 shows a short overview of the primary benefits and drawbacks associated with the presented models. The differences in their performance will be compared more thoroughly in the results and discussion section.

Models	Advantages	Disadvantages
MPNN	+ Uses a recurrent pooling layer which captures context	- Computation speed is bottle-necked by Set2Set (recurrent) pooling layer
Weave	+ Very fast computation and simple model + Can be used for edge information prediction	- Does not use powerful non-linear layers such as Set2Set or GRU
D-MPNN	+ Uses directional edge information + Fast computation	- Directional edge information often not available
MEGNet	+ Takes all forms of graph information into account (node, edge, graph)	- Slow - Requires useful graph level information (otherwise it is just noise)
AFP	+ Uses attention to filter out irrelevant information + Considers the graph on the node- and graph-level + Can provide insight into where attention is put and what nodes are deemed important	- Computation speed is bottle-necked by (recurrent) GRU activation layer

Table 1: An overview of the models implemented in the toolbox including some of the primary benefits and drawbacks. This table is not a direct comparison between models, but rather a representation of why one might use a model and the drawbacks associated with it.

3 Development of GraPE-Chem: Graph-based Property Estimation for Chemistry

The following section will cover the different features of `GraPE-Chem` and illustrate why they were implemented.

The initial goal of this project was to build a pipeline to make training GNN models straight forward while simultaneously allowing for direct control over the process. Inspired by the SciPy model pipeline (Virtanen et al., 2020), the implementation should be modular but complete for easy de- and reconstruction. Furthermore, it should be built on top of PyTorch-Geometric (PyG) (Fey and Lenssen, 2019), a versatile tool for graph representation learning with a lot of active development. A demonstration on how the toolbox can be used is in appendix E and F, and the online documentation is at (online-documentation). Figure 6 is a visual representation of the toolbox including some of the essential features as well as a pipeline of how it could be used.

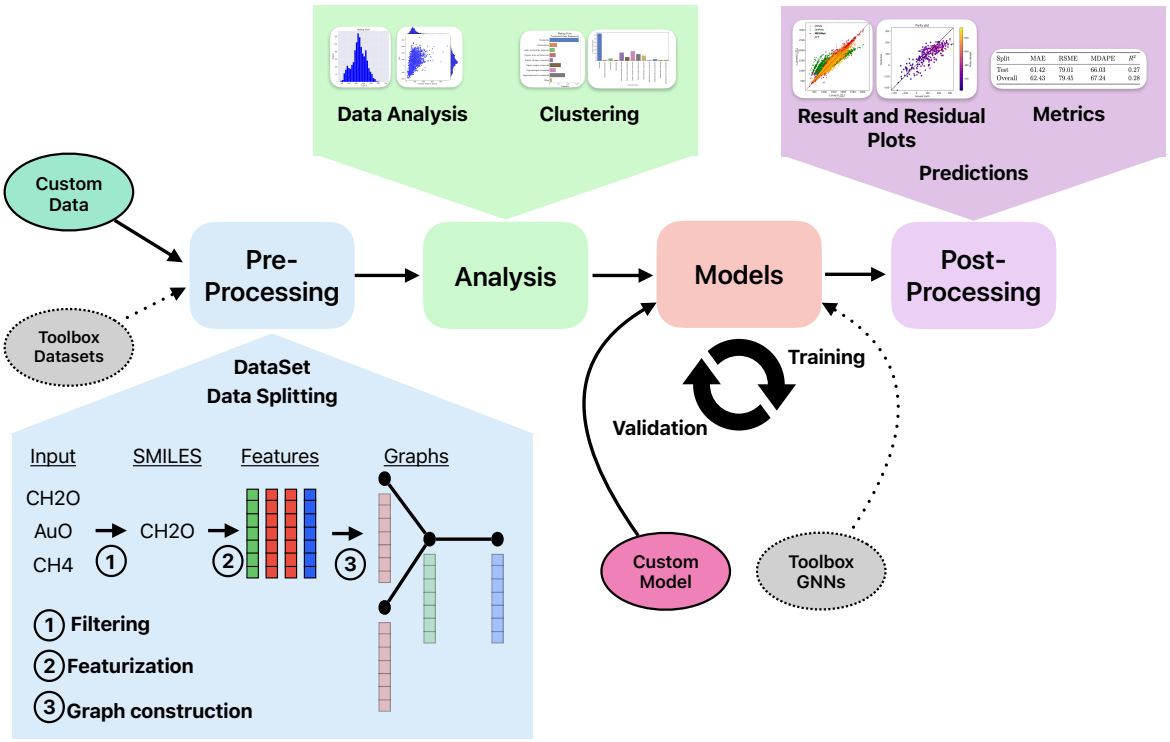


Figure 6: An overview of the `GraPE-Chem` toolbox consisting of the main features.

3.1 Pre-Processing

For graphs, loading the data is a non-trivial step in the machine learning process and especially so for chemical datasets. The molecules are stored as simplified molecular-input

line-entry system strings (SMILES; Weininger, 1988), and first need to be reconstructed into a usable molecular representation. For this step, the RDKit toolbox is used, in particular a function to transform a SMILES string into an object that holds the relevant information. To build the graph structure, the molecule will be split into its atoms, where each of the atoms will represent a node in the molecule graph. The atoms themselves and their bonds are then featurized to represent the relevant features as a single tensor. In detail, the preprocessing done by `GraPE-Chem` includes the following parts:

3.1.1 Filtering

As the first preprocessing step, the molecules are filtered based on three criteria: whether RDKit can reconstruct the molecule, whether the molecule consists of at least two heavy atoms and whether it only contains atoms that are 'allowed'. The last filtering option is often used to limit the dataset to purely organic compounds.

If any of the three criteria are not fulfilled, then the filter will discard that molecule. A lot of the functions in the preprocessing stage (like the featurizer) rely on RDKit functions, which in turn need a valid molecule representation. Filtering ensures that this requirement is always fulfilled.

3.1.2 Featurization

Graphs, or data to be fed to a GNN, essentially consist of four components: nodes and node features, as well as edges and edge features. To build molecule graphs, these fours parts need to be reconstructed from SMILES representations. Nodes are defined by the number of atoms per molecule, and the edges can be built given a sparse connectivity matrix of the molecule. Both of the information sets required for this are available through RDKit. But the features vectors are a bit more tricky. Here, information about an atom or bond is extracted and encoded as a vector. For an atom, the information itself could be a one hot encoding of it's symbol, the atomic mass, the formal charge, the number of hydrogen attached, etc. For a bond, the bond type and whether it is part of a ring might be interesting.

All of these features can be found using corresponding RDKit functions. The current featurization implementation uses classes inspired by DGL-LifeSci (Li et al., 2021), and lets the user choose the features that are encoded using a list. For instance, the following featurization instance could be called:

```
1 featurizer = AtomFeaturizer(allowed_atoms = ['C','O','H'],
    atom_feature_list = ['atom_type_one_hot', 'atom_degree', '
    atom_is_aromatic'])
2 featurizer(mol)
```

if the atoms were featurized using a one hot encoding of their symbols (with Carbon, Oxygen and Hydrogen as the 'allowed' atom types), as well as their degree and whether they are aromatic.

The same is possible with bonds:

```
bond_featurizer = BondFeaturizer(bond_feature_list = ['bond_type_one_hot', 'bond_is_conjugated']
bond_featurizer(mol)
```

to featurize based on bond type and whether the bond is conjugated.

Furthermore, to add more flexibility to the implemented features, there is an option to extend them with user defined functions. For example, using the `extend_features` method of the atom featurizer, another feature that checks if atoms are above some weight threshold could be added. Please see appendix E for a demonstration of this.

3.2 Data Classes

Inspired by the `Dataset` classes in PyTorch Geometric (Fey and Lenssen, 2019), the `GraPE-Chem` package uses `DataSet` and `GraphDataSet` classes for data handling. These classes are initialized with a list of smiles and target values among other various options, and create a dataset that a PyG GNN can use. Specifically, the smiles are first filtered, then featurized and finally saved as PyG `Data` objects which define a *graph*. The only difference between `DataSet` and `GraphDataSet` is that the latter can be initialized to split the data into training, testing and validation sets directly and inherits the rest from `DataSet`.

There are some points we wish to highlight concerning the implementation. For one, data management and analysis methods are implemented for the classes, such as saving the dataset (`save_dataset`), drawing individual smiles (`draw_smile`) and analyzing the dataset (`analysis`). There is also the option of initializing the dataset using *global* features, features that can be added at any point in the GNN, and have been shown to significantly increase the performance by themselves (Na et al., 2020).

This leads to the benefits of implementing and using these classes: On the one hand, they allow for easy and quick data management. On the other, they also let the user build on top the already implemented classes using python class inheritance with ease.

3.3 Analysis

There are several reasons why it is beneficial to analyse a chemical dataset before applying machine learning algorithms. For one, inspection of the compound groups contained in the dataset can reveal potential biases or uneven distributions. A model trained on mostly carbohydrates will not likely be good at predicting the melting point of organo chlorids. It can also help in determining a suitable clustering, such as Butina clustering to avoid heterogenous clusters (Butina, 1999), or data splitting algorithm based on traits such as molecular weight.

To assists in the analysis, they are several implemented data analysis tools, some of which are built into the `DataSet` class itself.

The primary of these functions is `smiles_analysis`, which uses DGL-LifeSci (Li et al., 2021) as an back-end to write an analysis report of the dataset, including the frequencies of atom symbols, bond, rings and more. If desired, it can also output the corresponding fre-

quency bar charts. This can be extended with the `mol_weight_vs_target` plot which draws the molecular weight distribution of the dataset against the target variable distribution.

3.4 Naive Classification

There are two separate ways of considering compound classes and their frequencies directly. The simple, quick but less informative way is using `classify_compounds` to read the SMILES strings and map each to a general class such as *Hydrocarbons* or *Oxygenated*. Note, that this is only a top level classification of the strings themselves and can be ambiguous.

3.5 Classyfire

The other, much more informative tools is using Classyfire (Djoumbou Feunang et al., 2016) through the `classyfire` function. The key idea is to use the extensive classification algorithm by Djoumbou Feunang et al. through their online application and then to retrieve the individual molecule information. In their advanced classification scheme, they define multiple layers of molecule classification, with each layer increasing the specificity of the class. The top level, *Kingdom*, splits the data into organic and inorganic compounds; the second layer, *Super-Class*, includes 26 organic and 5 inorganic categories and the number only increases exponentially with each deeper layer. This results in very informative and accurate classification, even in the top three layers of information, as seen in figure 7. The unfortunate downside is that the information retrieval will take around 6 to 9 minutes for 100 molecules, so for a large dataset with upwards of 100,000 molecules it could take a day or more. Furthermore, the algorithm primarily relies on (1) that the SMILES provided are correct and (2) that the information can be retrieved from the database. The latter especially tends to be a significant issue for generated or calculated datasets. `GraPE-Chem`'s `classyfire` includes some safeguards against incorrect SMILES, but it does not guarantee success.

3.6 Data Splitting and Clustering

After the data is fully loaded and the graphs are constructed, we can start to consider how to feed the dataset to a GNN. A key point here is how the data is *split* into the training, testing and validation sets. DGL-LifeSci (Li et al., 2021) is again used, specifically their implementation of the following splits: `Random`, `Consecutive` (using the input directly), `Molecular Weight` (which sort the molecules by weight and then splits), `Scaffold` and `Stratified`. To briefly explain, Scaffold splitting is based on the work of Bemis and Murcko, 1996, and uses a set of graph based rules to create scaffolds of molecules, frameworks that match each other. These scaffolds are then sampled uniformly to create the splits. And stratified splits come from group indicators passed to the function that are then also sampled uniformly to guarantee a fair distribution.

Furthermore, the options are expanded by introducing two splits based on **Butina** clustering (Butina, 1999). First introduced by Darko Butina, the idea is to add inductive bias

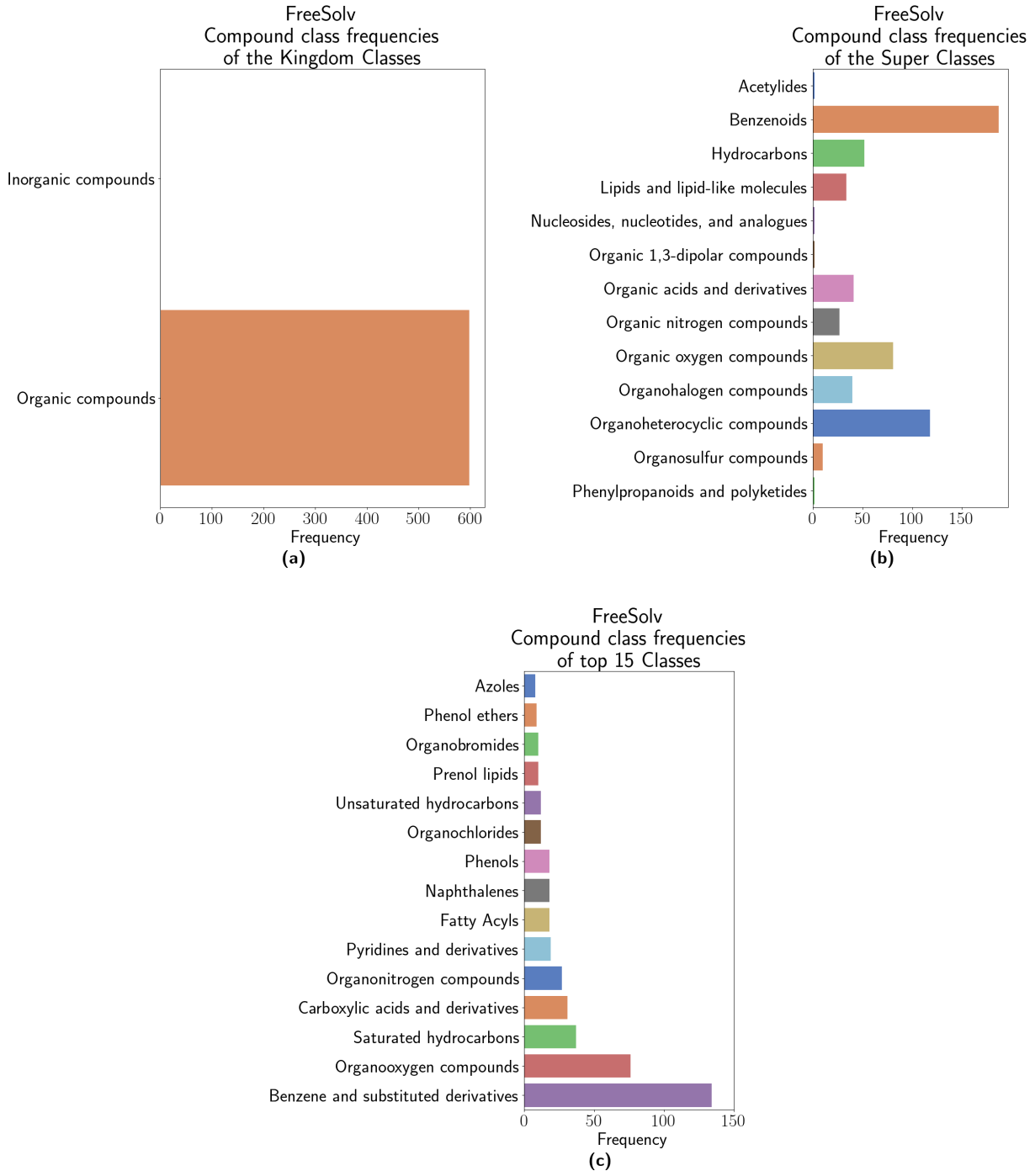


Figure 7: An overview of the first 3 classyfire layers applied to the FreeSolv dataset by Mobley and Guthrie, 2014. **(a)** shows the *Kingdom Class* layer, which only consists of organic compounds here. Note, that we filtered out any inorganic compounds first, so this plot is trivial. **(b)** shows the second, or *Super Class* layer. And **(c)** plots the frequencies of the top 20 most populated classes of the third layer, the *Class* layer.

by clustering molecules based on their similarity and prevent the formation of very heterogeneous clusters. The exact algorithm can be found in Appendix A. In practice, this algorithm is mostly implemented in RDKit, the only missing steps are to find the Morgan fingerprints ("Daylight Theory Manual", 2011) and calculating the pairwise Tanimoto similarity index (Tanimoto, 1958).

The first of the two split options is to just sample from the clusters uniformly, which ensures a fair representation across all splits.

In the other, clusters are sampled from largest to smallest and fill up the training, validation and test split in order. Using this split, the model will train on mostly large, homogeneous clusters and validate as well as test on the remaining small clusters or singletons. Martin et al., 2019, employed a very similar strategy, and according to them, splitting in this manner mimics the way chemists will propose novel, unique molecules for testing rather than well established ones. By using this variation of the Butina clustering split, large datasets can be used to emulate 'realistic' work environments (Martin et al., 2019).

3.7 Datasets

A key feature in modern packages are datasets like those found in PyTorch (Paszke et al., 2019) or PyTorch Geometric (Fey and Lenssen, 2019). They simplify the process of downloading and instantiating a dataset to a single line of code, often with useful transformation built in. Inspired by these, `GraPE-Chem` includes a few datasets itself, inlcuding the Jean-Claude Bradley Open Melting Point Dataset (Bradley et al., 2014), the LogP dataset (first introduced by Mansouri et al., 2016 and later corrected by Ulrich et al., 2021), as well as the QM9 dataset and FreeSolv dataset, both curated by Wu et al., 2017.

Like in PyTorch, these datasets will be downloaded automatically (should the code not be able to find the dataset in immediate file system). They are then cleaned as well as filtered and featurized by the `DataSet` class. These final three steps is what sets the `GraPE-Chem` implementation apart from the PyTorch Geometric implementation. All four datasets will be discussed more further down in the experiments section.

3.8 Models

As of writing, all of the implemented models are presented in table 1. Benchmarks of the existing ones will be given and discussed in sections 5 and 6.

All models in the toolbox can be used in two ways: they can either be used for embedding a graph as part of a larger model, or as full regression model outright. In the latter case, we use the GNN in question for embedding and add a model specific pooling layer as well as an input dependent MLP predictor.

Note that all models can handle the global features mentioned in section 3.2. For a brief presentation of how global features are implemented see figure 8, as well as appendix F, an advanced demonstration of GraPE. The key idea is, that the simple concatentation of a global feature has been shown to increase the performance of a model without addition of any new resources (Na et al., 2020). Furthermore, work like the S-GNN framework (Aouichaoui et al.,

2023b) prove that the addition of a correlated chemical feature can significantly increase the prediction quality of molecular properties.

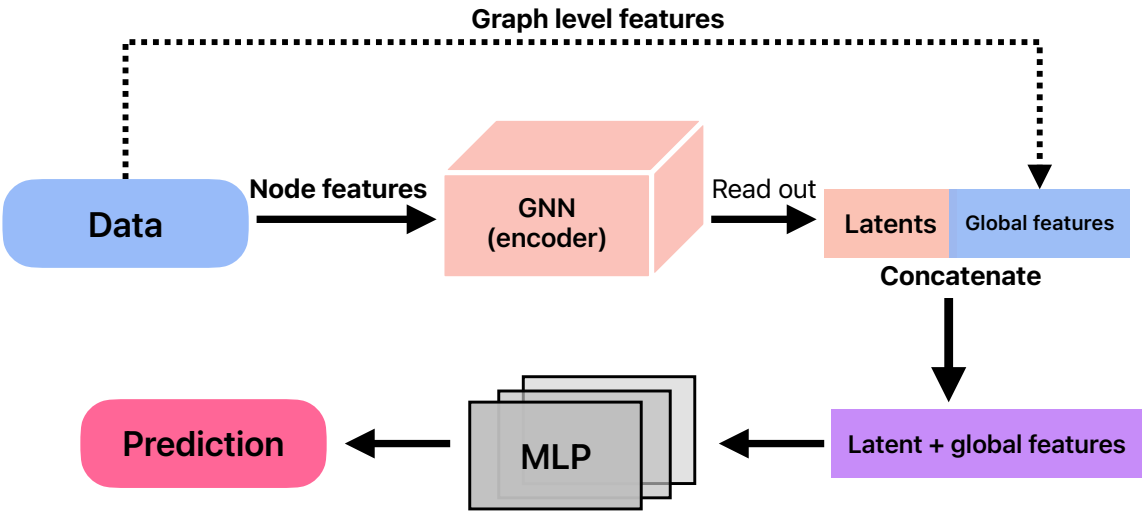


Figure 8: A visualization of how global features are added to the latent representation of a GNN. Note, that the global features are not treated before being concatenated to the latents (apart from standardization).

3.9 Post-Analysis

Like in any machine learning pipeline, some of the final steps concern the analysis of whatever the trained model produces. The toolbox includes functions to quickly generate key plots like the `Parity` plot, `Residual` plot and more. The use of all of these will be demonstrated in the Results section further down or in appendix E.

Another interesting avenue of post-analysis is to consider the latent representations produced by the model. For example, one could check if the last latent representation at the end of a GNN matches the expected behaviour of the data. This could be whether similar points are clustered together, and alternatively, one could use the latents to investigate trends in the output. Due to the high-dimensionality of the latents, as the hidden dimension is usually greater than 3, visualizing them can be tricky. Currently, the two tools are made available: the Principal Component Analysis, `PCA`, (Maćkiewicz and Ratajczak, 1993) and the T-distributed Stochastic Neighbor Embedding, `t-SNE`, (van der Maaten and Hinton, 2008). Both are based on the Sci-Kit Learn implementation (Pedregosa et al., 2011).

In a PCA, the orthogonal variance is extracted in the form of principal components indicating data trends. By plotting plotting the first two, potential underlying data clusters. The t-SNE, on the other hand, is an algorithm to group data points based their pairwise probability density. The use of both is demonstrated in the results section, 6.

3.10 Prediction

Lastly, after training and testing using a variety of post-analysis tool, the model can be used for prediction of new compounds. This process is simplified by giving the `DataSet` class a `predict_smiles` method that takes in SMILES and a model, featurizes the SMILES based on the `DataSet` instance, scales the output and returns a dictionary of each SMILES and their corresponding predicted value.

The key, here, is the fact that once a model is trained on a given dataset, any prediction will be based on the features and target property that the dataset was instantiated with. This is a necessary criteria for any prediction, so our implementation is just a simplification of this process.

4 Machine Learning Operations

This section summarizes some machine learning operations techniques used in the project and why they are important.

When developing and deploying packages like `GraPE-Chem`, one must consider a multitude of factors, including but not limited to building, deploying, monitoring, fixing and re-deploying. Although simplified, this cycle is referred to as DevOps (Development Operations) and it has been one of the primary guidelines for the modern computer-scientist. However, a new term specifically for the machine learning development cycle has been coined: MLOps. Like DevOps, it is concerned with the life-cycle of a package or software, specifically of a machine learning model that has to be (1) built, (2) deployed and (3) monitored.

There are several different interpretations of MLOps, like what is best practice, what is crucial and what is not, and more. Examining figure 9, the following key-words from the figure are key for machine learning: Modelling, Deploy, Monitor, Package, Reproduceability and Version Control. Possibly also Cloud, if a model is to work remotely and on request.

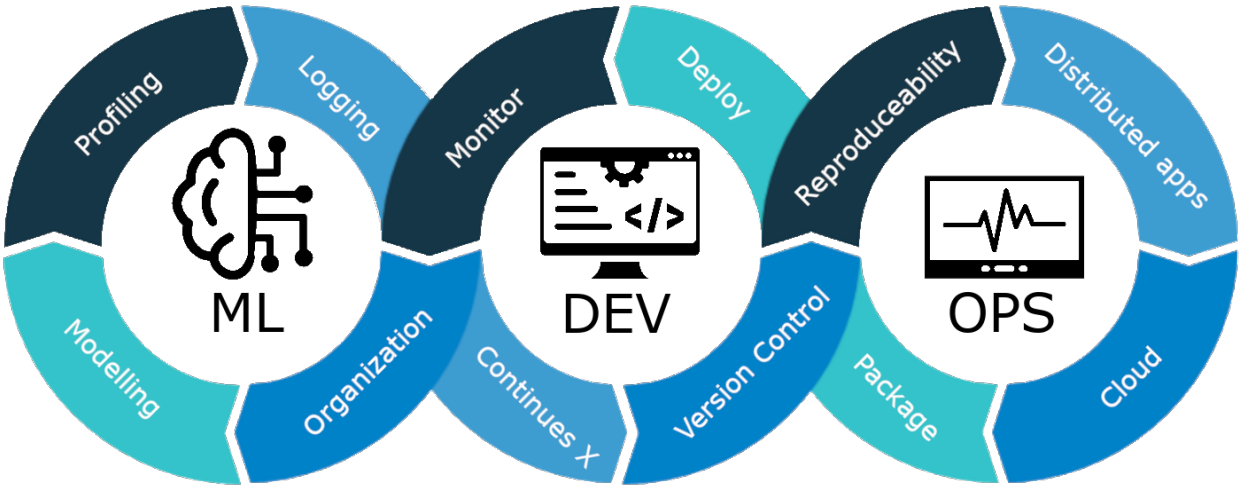


Figure 9: The cycle of machine learning operations (Detlefsen, 2024), represented by the primary parts: **ML** (modelling), **DEV** (development) and **OPS** (operations).

Unfortunately, following every MLOps step is not only time intensive, but usually requires extensive knowledge on several different aspects of development. So, staying within the scope of the project, it was decided to focus on modelling, package development and reproduceability. And within these aspects, the code documentation and the deployment of GraPE a package was the main focus. Additional future development avenues were considered, see section 8 for more details.

4.1 Code Documentation

One of the most important features of a code base is a good documentation. This is not only essential for the user, but provides readability for a developer. Throughout this project,

special care was given to writing clean code and good, detailed documentation. This includes extensive program and parameter overviews, marking each input with the correct data type for seamless use, and referencing sources within the documentation where used.

Furthermore, to allow the user to search and browse the documentation with easy, online documentation for the package was generated. Additionally, a general and advanced demonstration of the toolbox is provided online such that a new user can become familiar with the GraPE semantics.

4.2 Package and Deployment

To allow for fast and easy deployment of GraPE, PyPI or Python Package Index is used, an online repository for python packages. This means, that the user can install GraPE with a single command line, during which the required packages are installed automatically. Furthermore, with PyPI it becomes trivial to monitor the package for any required changes or bugs, and when necessary, re-deploy `GraPE-Chem`.

5 Experiments

The following section elaborates on the experimental setup used in this project. This includes a description of the datasets, what models were used and the hyperparameter optimization for said models.

5.1 Data

To investigate the performance of both the models implemented and the toolbox as a whole, all implemented datasets were used and are briefly described in the table 2. Note that standard filtering procedures were applied to the data, see section 3.1.1 for more information.

Shorthand	Property	Nature (Experimental or Calculated)	No. data [2]	Source
MP	Normal melting point (T_m)	Experimental and curated	2,989	Bradley et al., 2014
LogP	Octanol-water partition coefficient ($\log K_{ow}$)	Experimental, but corrected using calculations	5,526	Mansouri et al., 2016 Ulrich et al., 2021
Heat cap	Heat capacity (c_v)	Calculated (using DFT [3])	128,096	Wu et al., 2017
FreeSolv	Hydration Enthalpy (H_{hyd})	Experimental and Calculated	638	Mobley and Guthrie, 2014 Wu et al., 2017

Table 2: In order, the table describes the following physical properties: T_m is the melting point of a compound under 1 atm in degrees celcius; $\log K_{ow}$ is the log value of a chemical's concentration in the octanol phase to its concentration in the aqueous phase of a two-phase octanol/water system (Aouichaoui et al., 2023c); H_C is the required heat to increase a compounds temperature by one degree celcius; and H_{hyd} is the energy released by one mole of a compounds phase-shifting from vapor to water.

For the featurization step of pre-processing the data, the features from (Aouichaoui et al., 2023a; Aouichaoui et al., 2023c) were selected, see table 3.

[2] This is the number of data after filtering out in-organic compounds, compounds with less than 2 heavy atoms and ones containing atom types not considered, see table 3.

[3] DFT stands for density functional theory, and is a quantum mechanical modelling tool for electron structures. Using it, one can find numerical approximations of molecule and electron properties, see for example "What is Density Functional Theory?", 2009.

Feature (per node or edge)	Description	Encoding Type	Size
Atom features			
Atom type	type of atom (C, N, O, S, F, Cl, Br, I, P)	One-hot	9
No. of bonds	no. of attached bonds (0, 1, 2, 3, 4, 5)	One-hot	6
No. of H's	no. of Hydrogen attached to the atom (0, 1, 2, 3, 4)	One-hot	5
Valency	explicit valency (0, 1, 2, 3, 4, 5)	One-hot	6
Hybridization	hybridization (sp, sp2, sp3, sp3d, sp3d2)	One-hot	5
Aromaticity	whether the atom is in an aromatic system (0,1)	Binary	1
Chirality center	whether the atom is a chiral center (0,1)	Binary	1
Chirality type	type of atom chirality (R, S)	One-hot	2
Chirality tag	tag assigned to chiral center (unspecified, tetrahedral_CW, tetrahedral_CCW, other)	One-hot	4
Formal charge	the atom's assigned formal charge	Integer	1
Bond features			
Bond type	bond type (single, double, triple, aromatic)	One-hot	4
Conjugation	whether the bond is conjugated (0,1)	Binary	1
Ring	whether the bond is part of a ring (0,1)	Binary	1
Stereochemistry	bond stereochemistry (none, any, Z/E, Cis/Trans)	One-hot	6

Table 3: An overview of the features used for the featurization of the nodes and edges. The features used are the same as in (Aouichaoui et al., 2023a) and (Aouichaoui et al., 2023c).

Figure 10 shows the data distributions of the four datasets. Observe, that the Melting Point and the Log-P datasets are approximately normal distributed with centers around $300K$ and 2 respectively. The Hydration enthalpy is skewed slightly to the right, with many outliers in the negatives. This indicates that the hydration enthalpy is mostly be in the interval $[-10, 5]\frac{kJ}{mol}$. And lastly, the heat capacity is skewed slightly to the left, with a right heavy tail, and centered around approximately $1200\frac{cal}{molK}$.

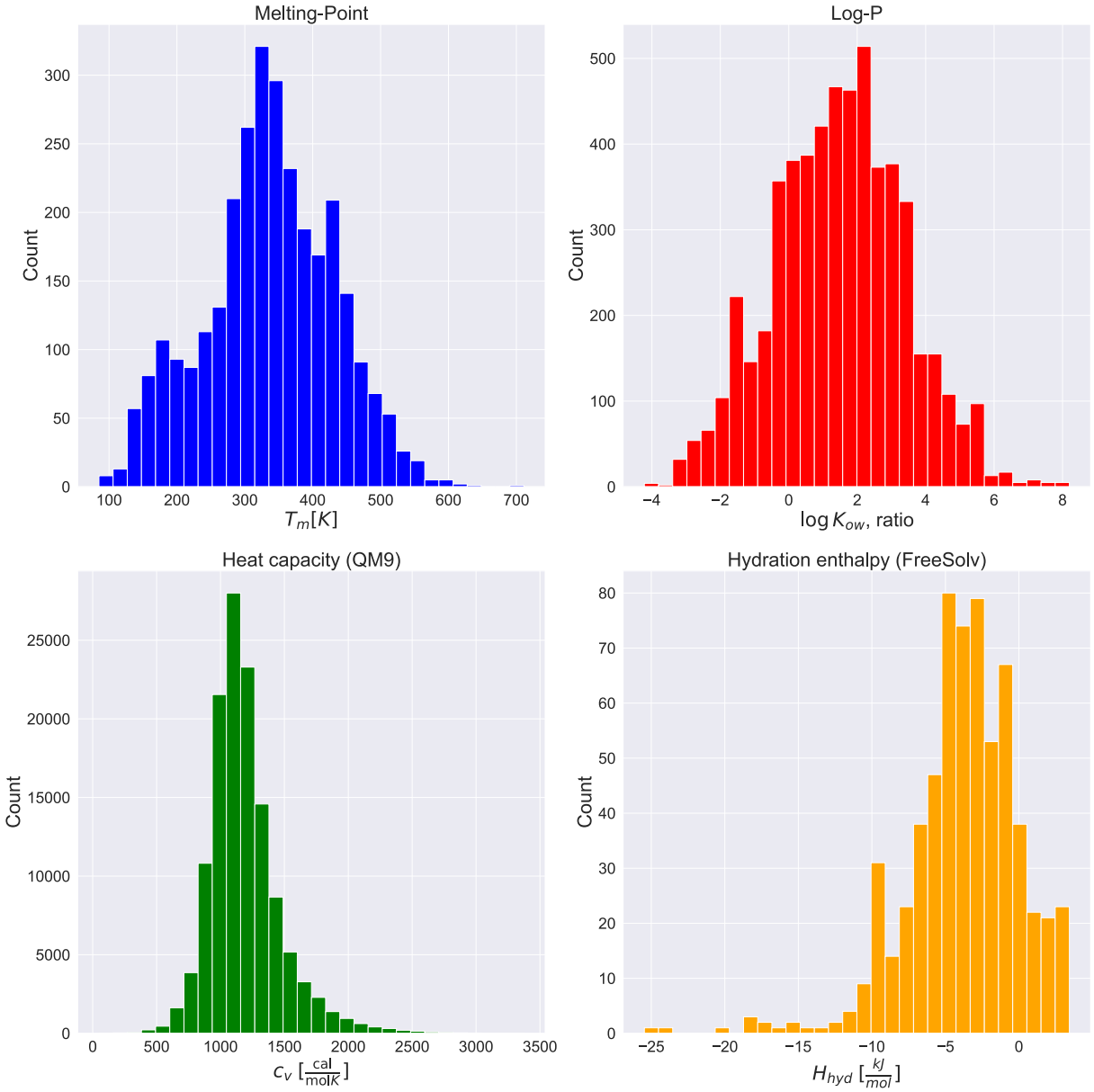


Figure 10: Histograms over all four datasets used for the experiments.

To investigate the compound distribution of the datasets, the previously mentioned `classyfire` function, see section 3.5, from the toolbox was employed. Figure 11 shows the second layer of the classification information for each dataset. According to the plot, all four datasets consist mostly of benzenoids, organoheterocyclic compounds, organic oxygen compounds and Hydrocarbons. Considering the allowed atom symbols, see table 3, and that inorganic compounds are filtered out, this is mostly expected.

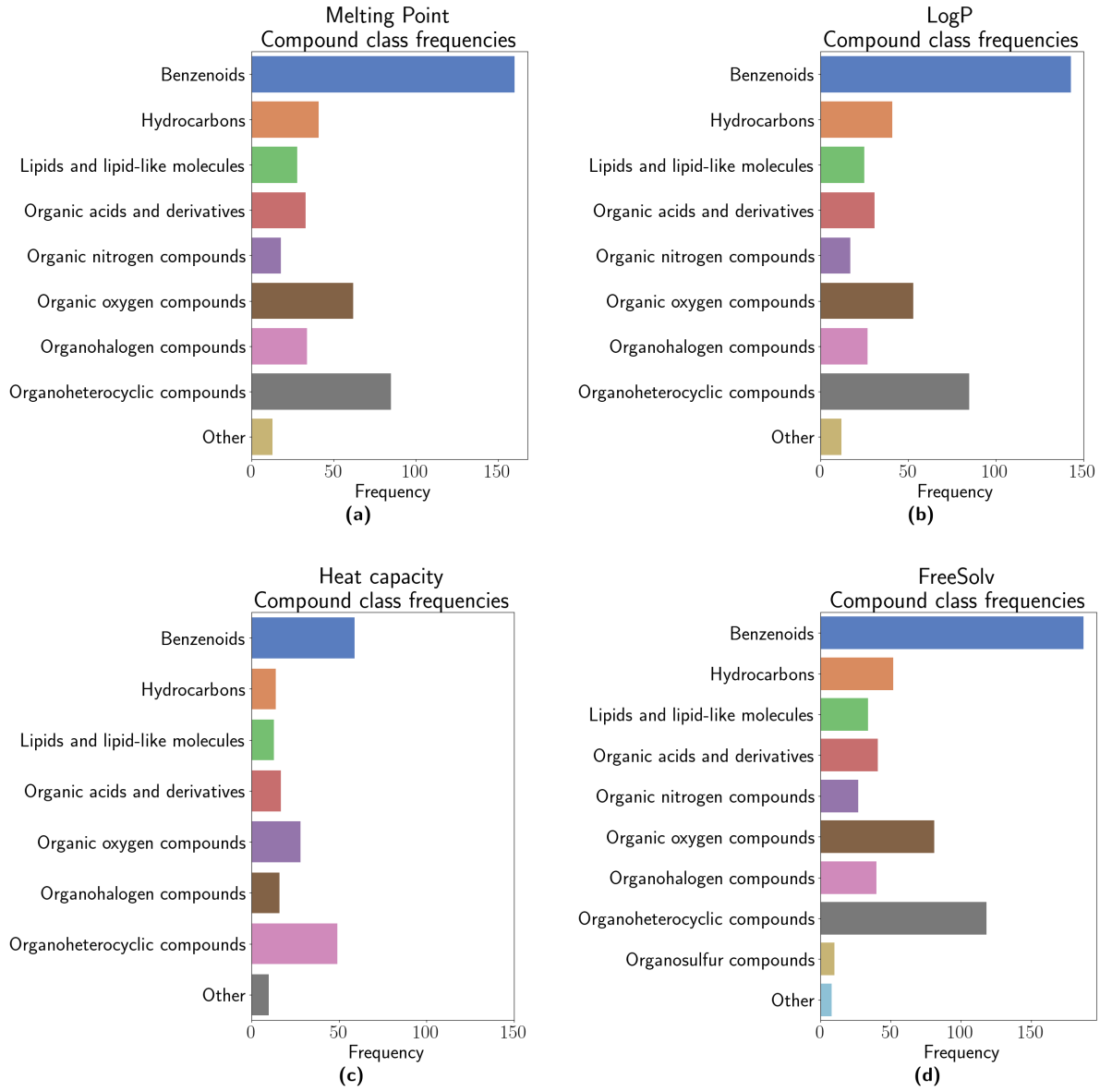


Figure 11: The classyfire algorithm applied to random subsets 0f 500 samples for each dataset. The information extracted here is the second layer, the *Super-Class* layer. (a)-(d) show the individual histograms. We observe, that all four datasets are evenly distributed in their classes. The only difference can be seen in (c), where we were not able to extract the same number of classes. This is likely due to the algorithm itself, please see 3.5 for more information.

5.2 Data Splits

For the experiments, the data was split randomly into 80/10/10 parts. The subsets will then be the training, validation and testing data respectively, the first two of which are used

in the optimization of the models and the last one is exclusively used for tests. The target properties are further scaled based on the training set and re-scaled for prediction. Although different splits were possible, see section 3.6, these were not further investigated and left for future work.

5.3 Model Selection

Based on the advantages and disadvantages presented in table 1, the Weave model was excluded from the main experiments of the report. This is due the simple architecture of the model, likely preventing it from performing as well as the others. Instead, Weave will be discussed and compared on the melting point dataset in appendix C. The remaining toolbox models, MPNN, DMPNN, MEGNet and AFP will be optimized, trained and tested.

5.4 Optimization and Hyperparameter Choices

The last aspect of the experimental setup are the hyperparameters of the selected models. A hyperparameter describes any choice made during modelling, such as the model itself, specific parameters thereof, optimizers, loss functions and more. For this report, the Adam optimizer Kingma and Ba, 2017 and the mean-absolute error function are used for training.

The remaining choices concern specific numerical values used in the models like the number of message passing layers. Optimizing these is crucial to achieving the best possible performance on a given dataset. The parameters chosen for optimization and their ranges can be found in table 4.To obtain the optimal parameter values, the Bayesian Optimization and Hyperband (BOHB) algorithm by Falkner et al., 2018, further described below, was employed. The exception to this are the MEGNet model parameters, and the reasons why are covered in section 7.5. Instead, a grid search over the same parameter space was used.

5.4.1 Optimization Algorithm

Here, the hyper-parameter optimization (HPO) paradigm itself is briefly discussed. The BOHB algorithm combined two different HPO approaches: Bayesian Optimization (BO; Shahriari et al., 2016) and bandit-based methods from Hyperband (HB; Li et al., 2018). The former, BO, leverages Gaussian processes to estimate the best combination of hyperparameters. This is done by evaluating the model with a sampled set of hyperparameters and fitting a Gaussian Process on the current and all past evaluations. The next sample from the process will then be an informed guess based on the past trials, allowing the algorithm to continuously improve the parameter guesses. Given enough time, BO will converge to a distribution of good hyperparameter values, but the convergence is often very slow (Falkner et al., 2018). To improve this, Falkner et al. added a Hyperband trial scheduler. The key idea of HB is to reduce time wasted on bad trials by using smart resource allocation. If a trial is promising it will be given time to finish, otherwise it will be terminated. The combination of both in BOHB creates a powerful and efficient HPO algorithm. In this report, the BOHB version implemented in the Ray-Tune (Liaw et al., 2018) package was used for all hyper-parameter optimizations.

5.4.2 Search Space

Parameter	Range
Global	
Initial learning rate	$[10^{-5}, 10^{-1}]$
Weight decay	$[10^{-6}, 10^{-1}]$
All Models	
Hidden (node) dimension	Integer($[32, 256]$)
Node embedding layers (T)	Integer($[1, 4]$)
Representation dropout	$[0, 0.4]$
MLP layers [4]	Integer($[1, 4]$)
AFP	
Graph embedding layers (L)	Integer($[1, 4]$)
MEGNet	
Hidden edge dimension	Integer($[32, 256]$)

Table 4: An overview of the hyperparameters and their search-spaces for HPO. The optimization results can be found in and the tables [6, 7, 8, 9] in appendix B.

The search space for all parameters, or the ranges used during optimization, can be found in table 4 and the optimization results in tables [6, 7, 8, 9] in appendix B.

The remaining parameters, the *learning rate reduction factor*, *patience* used for the early stopping, the number of *iterations* and the *batch-size* are set to 0.999, 30, 300 and 300 respectively. The first three choices were made based on trial-and-error, while the latter set to be as large as possible. Motivated by (Godbole et al., 2023), this choice reduces the stochasticity of training (Murphy, 2023) while exploiting all available hardware memory.

[4] The hidden units per layer are calculated as $64 \cdot 2^i$, where i starts at the last layer. For example, an MLP with two layers will have [128] and then [64] hidden units.

6 Results and Benchmarking

This section will cover the results from training the models based on their optimal hyperparameters (see tables [6, 7, 8, 9] in appendix B) using the same data-splits and testing various post-analysis tools on them. This includes metrics based on the test-set predictions, an investigation of the residual space as well as a brief latent space analysis. A sanity-check of the models based on the melting point prediction is used to examine how well the models extrapolate on new data, see appendix D.

6.1 Result Metrics

Like mentioned above, the complete result metrics together with some state-of-the-art (SOTA) comparison models can be found in table 5. The specific metrics are the mean absolute error (MAE), median-absolute percentage error (MDAPE), root-mean-squared error (RMSE) and the R^2 value. Note, that each metric represents a different prediction quality, and no one of them is inherently more correct than another. For example, the RSME punishes large outliers more than the MAE due to the square over the difference.

The following observations can be made from the table. The toolbox models perform worse than the comparison models on three datasets, but better on the LogP dataset. (Our) DMPNN is consistently either the worst or second worst model out of the `GraPE-Chem` models, while the other have dataset depended performance. MEGNet, for example, performs poorly on all but the heat capacity dataset. The general performance on the heat capacity dataset is poor compared to the SOTA models, even between the two DMPNN models. The toolbox LogP models could be state-of-the-art in performance, but this needs to be confirmed with additional studies. Furthermore, the original AFP authors achieved a better RSME than our AFP did. This is a curious result, and will be further examined in the discussion.

Overall, it is clear that the experiment models do not perform as well as the current SOTA models on the popular datasets. Interestingly, the performance difference is not large for small datasets but orders of magnitude apart for the large heat capacity dataset. Further points will be discussed below.

Dataset	Model	Nr. params	MAE		RMSE		MDAPE		R^2	
			Test	Overall	Test	Overall	Test	Overall	Test	Overall
T_m	MPNN	149,031	31.30	28.87	41.01	37.36	27.74	24.85	0.80	0.84
	DMPNN	23,439	33.57	30.29	42.18	39.50	34.48	30.17	0.79	0.82
	MEGNet	419,553	36.55	32.10	46.78	42.48	34.39	29.56	0.74	0.79
	AFP	24,573	31.18	28.92	41.99	38.93	29.34	27.88	0.79	0.83
	AGC [5]	2,679,351	26.90	21.80	36.30	29.30	26.40	20.60	0.86	0.91
	GroupGAT [5]	8,393,194	26.60	17.40	35.70	23.90	26.60	16.40	0.87	0.94
	E-MPNN [5]	1,280,321	28.60	22.00	39.80	29.80	27.10	21.00	0.82	0.90
	E-DMPNN [5]	116,965	30.00	25.80	39.80	34.40	29.20	25.00	0.82	0.87
	E-AFP [5]	783,235	26.90	22.30	37.80	29.90	25.66	21.00	0.84	0.90
LogP	MPNN	437,727	0.28	0.25	0.43	0.38	13.00	11.00	0.95	0.96
	DMPNN	33,499	0.41	0.38	0.64	0.58	17.81	15.85	0.88	0.90
	MEGNet	722,401	0.24	0.19	0.48	0.38	5.87	4.72	0.93	0.95
	AFP	76,777	0.17	0.17	0.29	0.29	5.65	5.77	0.97	0.97
	MLP [6]	N.A.	0.37	N.A.	0.51	N.A.	N.A.	N.A.	0.89	N.A.
Heat cap	MPNN	4,651,817	37.33 (0.14)	36.50 (0.14)	45.66	63.86	1.73	1.70	0.95	0.95
	DMPNN	94,989	99.27 (0.39)	98.38 (0.39)	153.43	152.24	4.60	4.57	0.70	0.70
	MEGNet	85,185	26.85 (0.10)	25.15 (0.10)	45.66	42.60	1.37	1.30	0.97	0.98
	AFP	22,195	56.64 (0.20)	56.36 (0.20)	83.13	83.04	3.61	3.59	0.91	0.91
	UNI-Mol [7]	N.A.	N.A.	$(4.67 \cdot 10^{-3})$	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
	D-MPNN (orig.) [8]	N.A.	N.A.	$(8.14 \cdot 10^{-3})$	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
FreeSolv	MPNN	406,119	0.85	0.70	1.30	1.05	14.87	14.75	0.82	0.90
	DMPNN	36,829	0.76	0.84	1.11	1.22	14.53	17.41	0.87	0.87
	MEGNet	85,185	1.03	1.01	1.45	1.56	22.00	19.01	0.78	0.80
	AFP	368,049	0.54	0.55	0.74	0.82	9.68	9.80	0.94	0.94
	AFP (orig.) [9]	N.A.	N.A.	N.A.	N.A.	0.74	N.A.	N.A.	N.A.	N.A.

Table 5: A compilation of all the results generated, with the metrics considered being the MAE (mean absolute error), RMSE (root mean square error), MDAPE (median absolute percentage error) and the R^2. Note, that the 'E' in for example E-MPNN stands for Ensemble and for certain values, the standardized valued are given in brackets. ([5] Aouichaoui et al., 2023c), ([6] Win et al., 2023), ([7] Zhou et al., 2023; Lu et al., 2023), ([8] Yang et al., 2019; Heid et al., 2024), ([9] Xiong et al., 2020)

6.2 Parity and Residual Density Plots

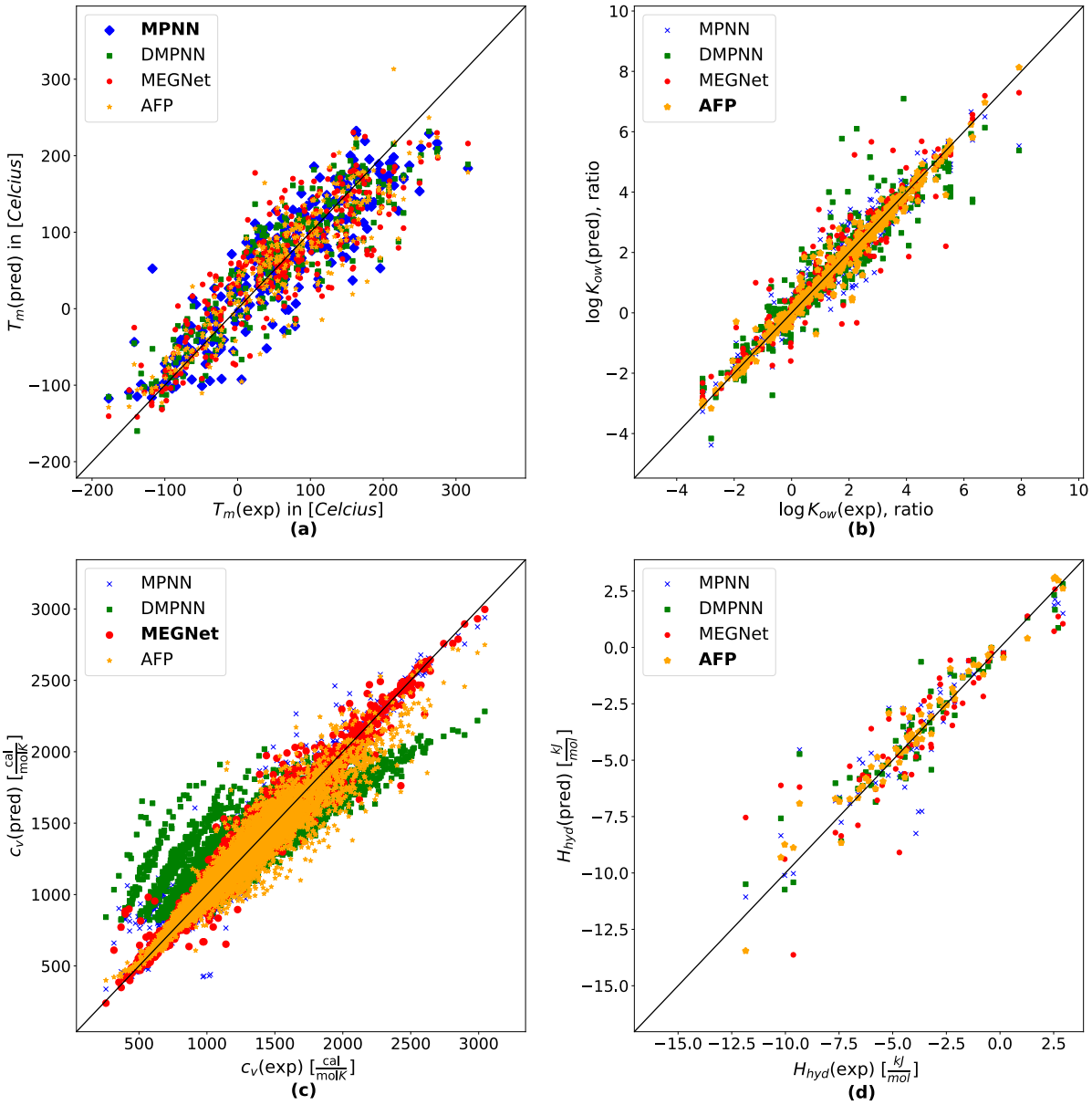


Figure 12: The parity plot of all four model predictions per test set, with the experimental values on the x-axis and the predicted ones on the y-axis. In each plot, the best model (based on the R^2 value, see table 5, is marked with bold font in the legend.

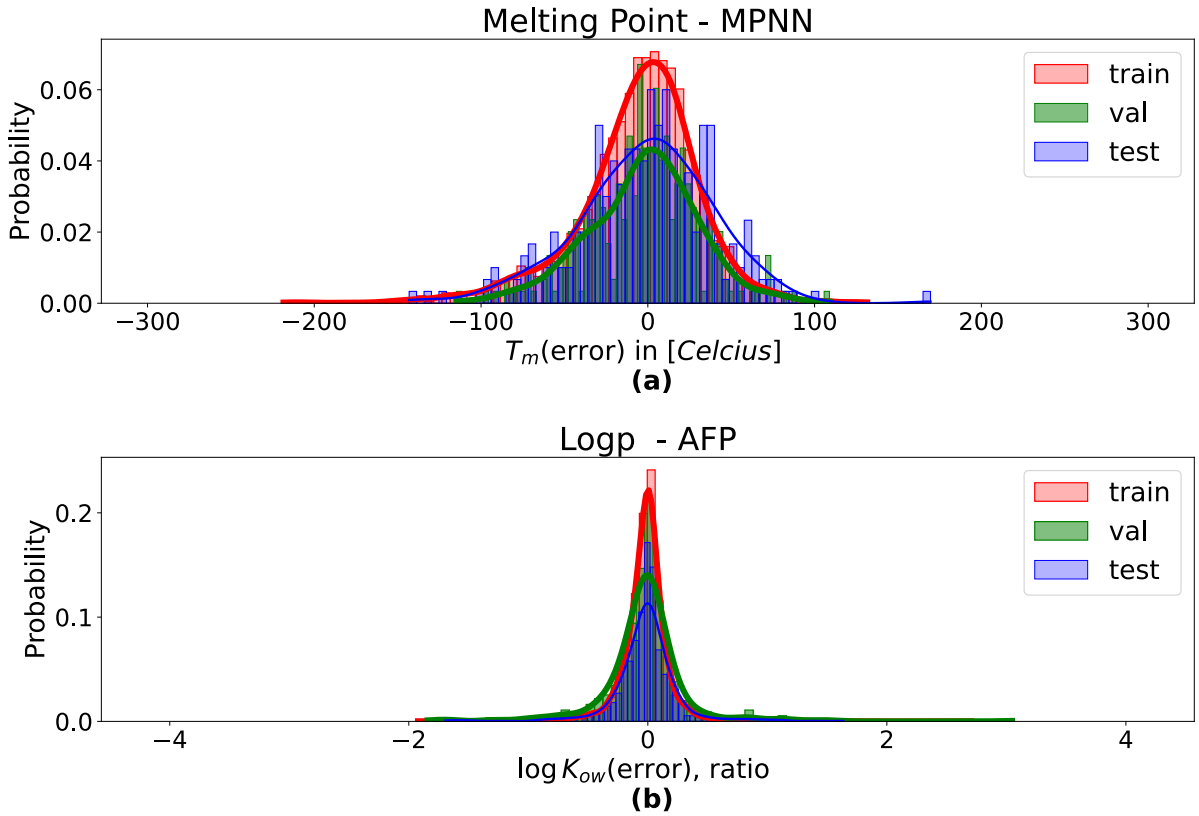


Figure 13: The residual density plots of each dataset and their best model (based on the R^2), consisting of: **(a)** the melting point and MPNN, **(b)** the log-p and AFP. *(cont.)*

In addition to the metrics table, the parity plots of each test set are presented in figure 12. Visually, all four models seems to perform similarly in 12**(a)**, which is supported by their metric being generally close. For the other plots, 12**(b)**-12**(d)**, there are some visual clues on which of the models is best. For 12**(b)** and 12**(d)** AFP is clearly the best, while in 12**(c)** MEGNet and AFP perform similarly.

For each of the best models, the density distribution of the residual is plotted in figure 13. This type of plot is usually used to check how the errors behave, i.e. if there are any non-linearities present or if the errors are badly distributed. If there are, that means there is an issue in the models. From all four plots (13**(a)**-13**(d)** the residuals are normally distributed, indicating that there are no problems within the modelling.

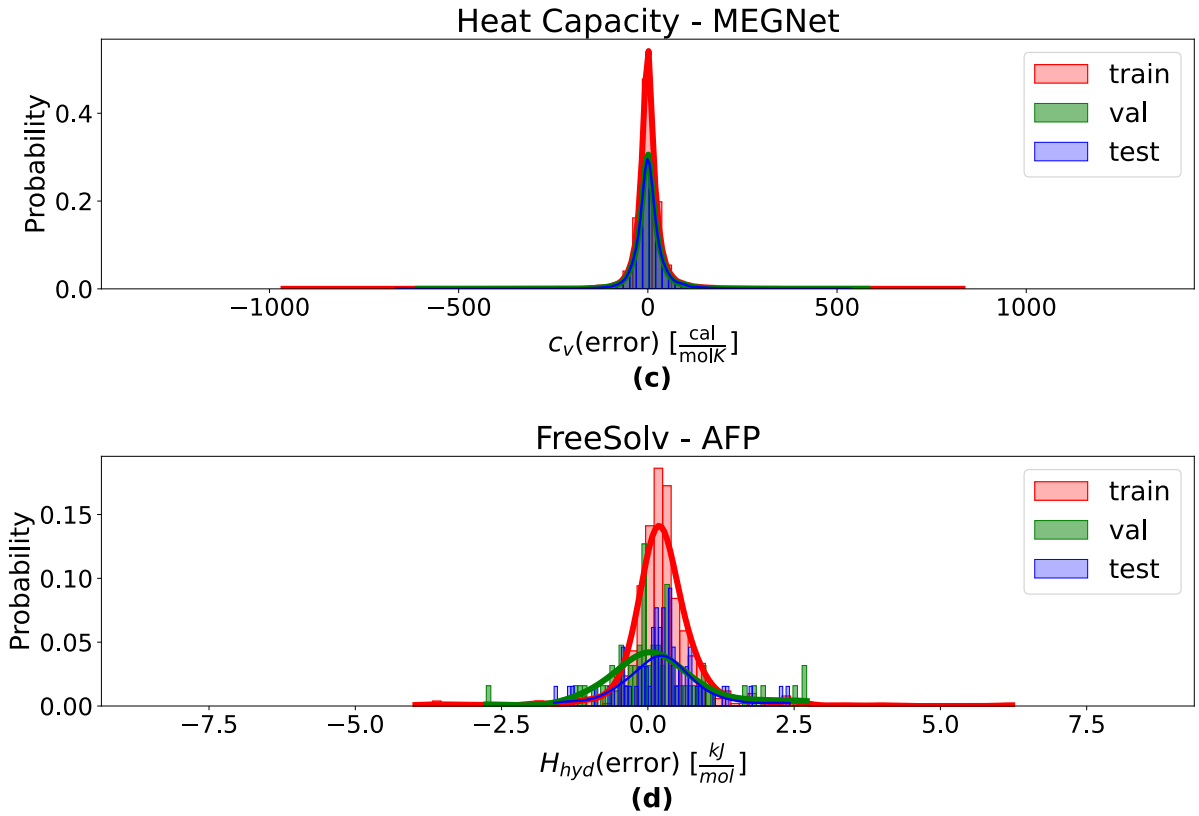


Figure 13: Continuing, **(c)** the heat capacity and MEGNet and **(d)** the free solvation energy and AFP.

6.3 Latent Space Analysis

In an effort to provide further insight into model performances and their domain of applicability, the last latent representations of the GNN models are examined. In this subsection, only the melting point dataset is analyzed, but the tools are implemented and available so more can be done in the future. Only the best performing model based on the R^2 value is considered, here the MPNN, and two types of latent space analysis tools are used: the Principal Component Analysis and the t-distributed stochastic neighbor embedding. The data points are labelled using the Classyfire procedure and each of their super-class.

6.3.1 PCA

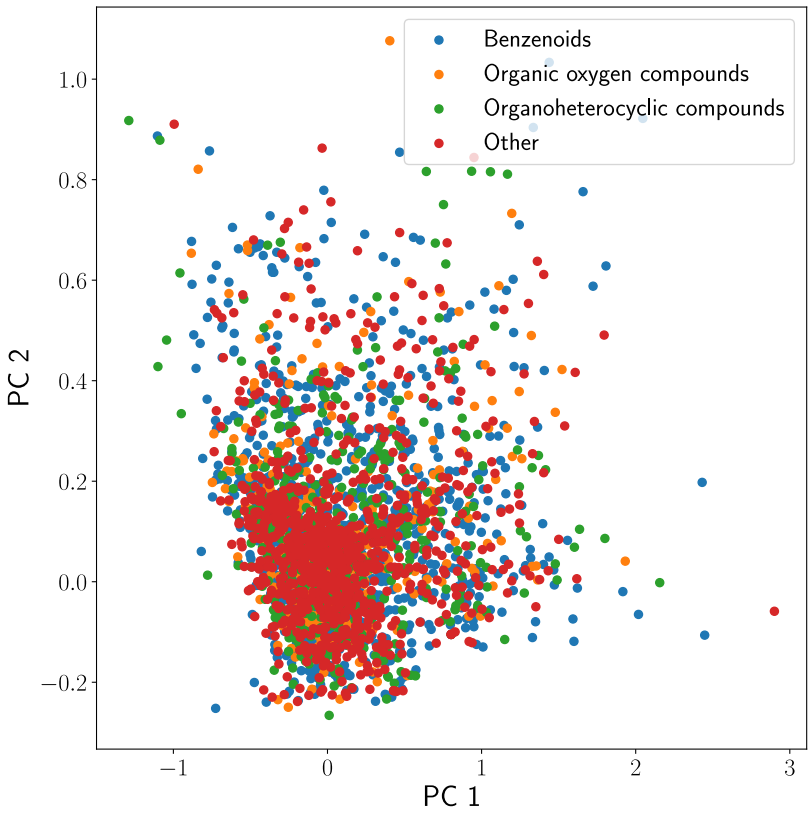


Figure 14: A figure showing the PCA plot of the latents extracted from the MPNN model predicting on the entire Melting Point dataset. Note that only the top 3 classes are given distinct colors while the rest are under 'Others'.

Applying the PCA and plotting the first two principal components in figure 14, no structure becomes visible. The reason for this could be that the dataset is homogeneous and mostly consists of similar compounds.

6.3.2 t-SNE

Lastly, the `t-SNE` procedure is applied and the first two components can be found in figure 15. Two clusters seem to exist, but the compounds themselves are mixed between them. This could indicate that there is another factor inducing the clustering, such as the molecular weight or a different property. However, further analysis is left for future work.

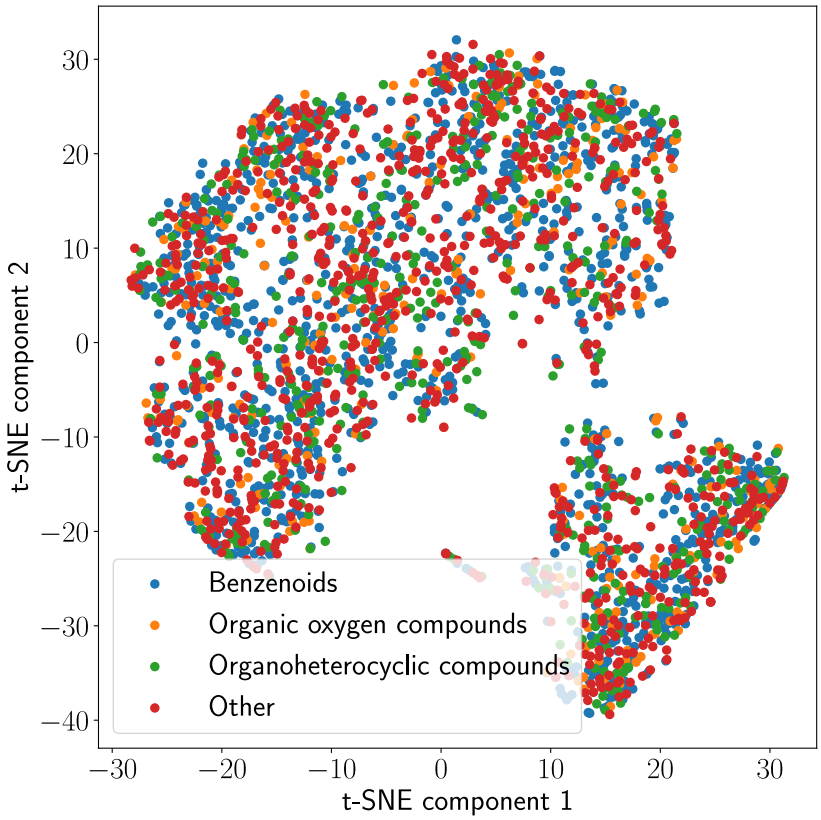


Figure 15: A figure of a two component t-SNE Embedding of latents extracted from the MPNN model predicting on the entire Melting Point dataset. Like in figure 14, only the top three most frequent classes are given a distinct color. The perplexity, approximately a cluster size measure (van der Maaten and Hinton, 2008), used was 100 and the number of iterations was 1750

.

7 Discussion

In this section, the implications and interpretations of our findings are considered, as well as some analysis of the limitations present in this project.

7.1 Metric Considerations

A significant issue when conducting the evaluation of the experiments were the comparison values from other papers. Especially for FreeSolv, it was not clear how the SOTA papers calculated their RSME values, whether they calculated it based on standardized values or not, and on what data split. Ideally, the authors should have been contacted and the exact data splits and evaluation would have been replicated for a correct comparison. Unfortunately, this issue only revealed itself too late, but this is going to be a crucial detail in future work.

7.2 Model Results and Future Improvements

Given the results from section 6, it is clear that more powerful models such as Uni-Mol or AGC generally outperform the implemented models on most datasets. Adding tools to build these models or implementing them directly would boost the overall performance of the toolbox. Furthermore, the addition of an ensemble framework could significantly improve the reliability and effectiveness of the already existing models.

The results also indicate that the models achieve good performance on small to medium sized datasets. In the field of chemistry, generating a large dataset is non trivial and expensive, which is why a small dataset is certainly possible in some settings. So achieving good results on smaller datasets is still a very relevant goal, and proves `GraPE-Chem` to be useful.

7.3 Training Time and Model Complexity

Another aspect worth considering in regards to model comparison is the training time and model complexity. Although the results were obtained in an academic setting where resource use is generally not a concern, if `GraPE-Chem` is to be used industrially or with limited resources, the performance in relation to how expensive it is to run has to be investigated.

Based on table 5 in section 6, the best AFP and DMPNN models tend to have less parameters than MPNN and MEGNet by an order of magnitude while still performing similarly or even better. Part of this is due to the way MPNN and MEGNet models are constructed, i.e. they use a lot of MLPs as update functions or in-between. The other reason is the recurrent set2set layer of MPNN which can quickly become a bottleneck. So under resource limitations like memory, AFP and DMPNN should be the first choice.

A similar argument can be made when examining the relative training time of each model in figure 16. DMPNN is the fastest, usually followed by AFP. The differences in speed are due to the set2set readout functions used in AFP and MPNN, and the general overabundance of MLPs in the MEGNet model. Note, that DMPNN performs very well if the complexity and run time are additional criteria. If the run time difference is not an issue, then the AFP model is overall the best.

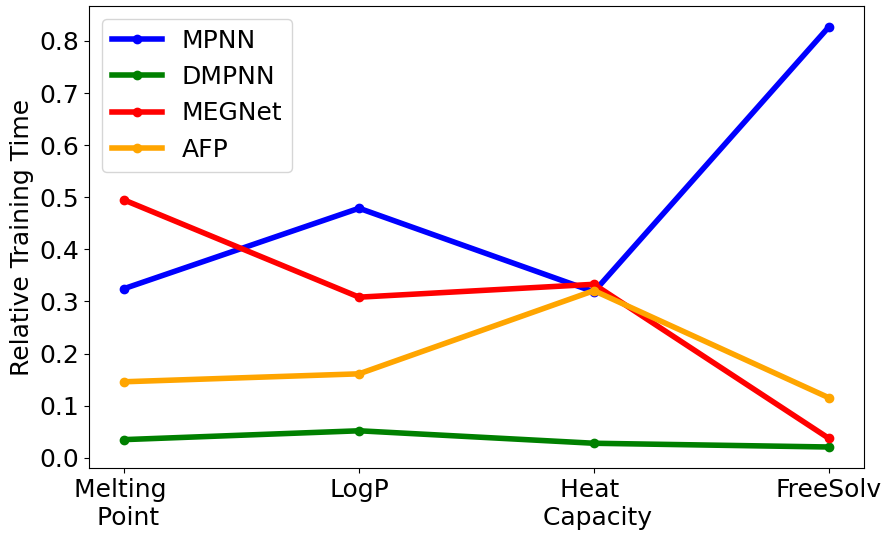


Figure 16: The relative training time of each model and dataset combination. Note, that this is a purely qualitative comparison.

7.4 Effects of Randomness on Performance

Although model complexity certainly has an effect on the difference in performance, certain random aspects can as well. How the data is split is especially important, as a good split can produce a (metrics-wise) much better model than a bad split. The difference between the toolbox AFP evaluations on FreeSolv and original AFP are likely due to the way the data is split. Because it is prohibitively expensive (or impossible) to test all data-splits, researchers often settle on a random one and then perform optimization based on it. If the data-split has good properties that support model training, it can lead to better overall performance than if not.

Other factors such as the random seeds used for the pseudo-random number generators during optimization and training can have an effect as well. For example, model initialization has a large impact on what local optima is found during training and whether it is the global optima. Another example is the combinations of hyperparameters tested during optimization, because if the globally optimal combination is found is also partly up to chance.

7.5 Hyperparameter Optimization Considerations

As mentioned briefly in the results above, some significant issues during the hyperparameter optimization of MEGNet were encountered. Specifically, the quality of the random samples (hyperparameters) drawn generally resulted in poor models. Even the optimal choices resulted in worse models than hand-tuned hyperparameters. We suspect there are three possible reasons why this happened: (1) the number of samples used during optimization

were too small, therefore the Bayesian optimization algorithm was not able to converge; (2) the search space was too broad, and as such the number of samples was insufficient; or (3) the optimization script itself was not correctly set up.

To stay within the time frame of this project, grid-search over selected points in the search space was used, but these measures did likely not result in the best possible models. This should be considered when interpreting the results, especially between the models optimized with BOHB and those that were not. Especially in regards to the heat capacity results, where MEGNet outperformed the other models with the inferior optimization scheme. This indicates that the performance can be pushed even further with better and optimized parameters.

8 Perspective and Future Work

This section will consider some aspects that were out of the scope of the project, but are interesting avenues for future work as well as a brief reflection on the goals of the project.

8.1 Future Model Development

As of writing, GraPE-Chem contains five models, all of which are used for chemical property prediction. But as mentioned before, the toolbox needs more powerful models to handle large datasets. Based on the discussed models so far, the mentioned FraGAT (Zhang et al., 2021), AGC (Aouichaoui et al., 2023a) or GroupGAT Aouichaoui et al., 2023c models could be implemented. All of these use the graph attention and fractions of compounds, allowing for in-depth analysis of the prediction process.

A different path would be adding models using geometric information for better predictions. This approach to molecular property prediction is currently state-of-the-art, proven by the before mentioned Uni-Mol (Zhou et al., 2023; Lu et al., 2023) model. Other examples are the MEGNet (using 3D information), the 3D infomax model (Stärk et al., 2022) and TensorNet (Simeon and de Fabritiis, 2023). Computer-vision tools like diffusion have also been shown to be effective for graph neural networks (Gasteiger et al., 2019; Chamberlain et al., 2021; Li et al., 2024), demonstrating that treating compounds as 3D objects rather than just graphs is a promising path.

Lastly, we could consider broadening the development goals to also include drug discovery by way of molecule generation. Here, the focus is on learning to build a model that can encode the graphs of interest, and then use the same or a coupled model to generate random graphs. The idea is then, that some of these graphs are useful and unique. Although this is a different area of research than pure chemical property prediction, it could prove to be an interesting extension of `GraPE-Chem` in the future. Promising works are, for example, the original graph variational auto-encoder (GVAE) (Kipf and Welling, 2016) which uses variational inference to build an effective generation model; the GRAN model (Liao et al., 2020) using auto-regressive conditioning for better graph generation; as well as the junction-tree variational auto-encoder (Jin et al., 2018) which combines chemical structure scaffolding with GNNs for their generation model.

8.2 MLOps - Package Development

Like mentioned in section 4, MLOps includes a lot of different possible avenues of development. The next two subsections will be used to document some points that will focused on in future work.

One tool that is essential for long term development is *unit tests*. They allow a package to be developed by multiple people at once without risking code inconsistencies. The most popular way of writing these is using GitHub and forcing automatic tests for new code that is being added to a project. That way, it ensures that the development on the package is mostly

or entirely productive by rejecting any code that does not pass the unit tests. While they could not implemented in the allocated time, they are crucial for long term development.

Another important part of a well performing python package is some form profiling tool. While python already has good built-in profilers such as `cProfile`, it is usually used for after-the-fact analysis. Packages like `Ray Tune` (Liaw et al., 2018) that deal with computation heavy processes sensitive to bottlenecks have built in, pro-active 'alarm systems'. Should a process stall or take an unreasonable amount of time, then the program will alert to that fact. If `GraPE-Chem` is to be used like that in the future, it needs similar tools to make it more problem resistant and easier to debug.

8.3 MLOps - Model Deployment

How the package is developed is not the only thing we need to consider for long term use. Another important question is how the package, and specifically the models within, can be deployed. Essentially, how can a potential user plug in a SMILES and get a prediction of some attribute without implementing the entire machine learning process.

Practically, we would have to combine all the elements of the current pipeline into one function or script that takes a SMILES as input as well as the prediction property and returns a value. Theoretically, this could be done by first using the `DataSet` loader (sec. 3.1) to featurize the SMILES input and generating the corresponding graph. Then a standard or specified, pre-trained model is loaded from a cloud storage container and the graph is passed through it. The model has to have been trained on the data that corresponds to the prediction type, example given, a model is trained using heat capacity data to predict the heat capacity. Finally, the prediction is re-scaled using the data-dependent values. All of these steps need to be executed sequentially inside of one executable. Projects like Chemprop (Yang et al., 2019; Heid et al., 2024) already have pipelines like the one described in place, and have proven that it is an effective way of making their tools practical.

A different, equally relevant deployment objective is the use of *Docker*. To give a brief introduction, Docker is an open source tool that lets users create 'containers', essentially virtual environments that hold all the information required to run the tool. Especially machine learning packages like `GraPE-Chem` need a lot of prerequisites to function as intended, and Docker has proven to be an effective way of guaranteeing those conditions. For a fully fleshed out toolbox, we would create a Docker 'image', a snapshot of all the required packages, and combine it the easy prediction pipeline discussed above. This would allow user to just install our package 'image', and then they can immediately start using it for predictions.

8.4 Project Reflection

The aim of this project was to explore Graph Neural Networks and build a machine learning pipeline for molecular property predictions. The code used for said pipeline should then be structured and documented well and make use of MLOps.

The code required for a toolbox of this scope was more than expected, and a key challenge was to focus on the foundation rather than small details in the beginning. As the project progressed, the choice was made to focus on models rather than datasets and data analysis

features which provided the time to properly implement them. At the end, however, the time required to setup hyperparameter optimization and run a large search space was underestimated, the reason why MEGNet had to be finished using grid-search. Time management in MLOps is difficult, but valuable experience was gained from this project.

Overall, some of the success were the detailed code documentation as well as publishing it online, the detailed experiments conducted and releasing `GraPE-Chem` as a package. If building a toolbox from scratch seemed impossible before the project, it does not anymore. The coding practice and knowledge picked up will also prove to be useful in future projects.

9 Conclusion

This report introduced `GraPE-Chem`, a toolbox for graph-based property prediction for chemistry. Furthermore, the five models implemented in the package were reviewed as well as the features of the package. The experimental setup is detailed and the hyperparameter optimization paradigm is summarized. The experiments conducted on four common chemical property datasets as well as the provided notebooks in appendices E-F showcase what the toolbox is capable of. Finally, a perspective over possible future development avenues such as model deployment and other features plus a reflection on the project is given.

The experiments showed, that while the toolbox is streamlined, the models currently implemented fall behind in performance compared to newer, more advanced models for large datasets. These models, however, can easily integrated into the toolbox in the future. Additionally, the toolbox provides sophisticated filtering and featurization tools that are customizable. Data analysis tools such as the pre-built plotting functions and Classyfire allow for deep insight into the compound distributions and their classes. Butina clustering and a variety of other split functions can be called from a `DataSet` object and let the user quickly try different splits. And lastly, the dataset specific prediction tool serves as a tractable way of testing any trained model.

In conclusion, while the models in the toolbox leave room for improvement, the toolbox itself promotes fast development iteration with a variety of data analysis tools. Currently, additional development related to this toolbox includes hybrid-models that leverage physics for more accuracy and interpret-ability, as well as an expansion of the currently available models with more advanced ones.

This, and any future progress from this toolbox will hopefully be able to make a positive impact in the space of chemical property prediction and potentially drug discovery.

References

Jones, A. W. (2011). Early drug discovery and the rise of pharmaceutical chemistry. *Drug Test. Anal.*, *3*(6), 337–344.

Desborough, M. J. R., & Keeling, D. M. (2017). The aspirin story – from willow to wonder drug. *Br. J. Haematol.*, *177*(5), 674–683.

Vamathevan, J., Clark, D., Czodrowski, P., Dunham, I., Ferran, E., Lee, G., Li, B., Madabhushi, A., Shah, P., Spitzer, M., & Zhao, S. (2019). Applications of machine learning in drug discovery and development. *Nature Reviews Drug Discovery*, *18*(6), 463–477. https://doi.org/10.1038/s41573-019-0024-5

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, *20*(1), 61–80. https://doi.org/10.1109/TNN.2008.2005605

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985, September). *Learning internal representations by error propagation* (tech. rep. No. ICS 8504). Institute for Cognitive Science, University of California. San Diego, California.

Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks.

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., Bridgland, A., Meyer, C., Kohl, S. A. A., Ballard, A. J., Cowie, A., Romera-Paredes, B., Nikolov, S., Jain, R., Adler, J., ... Hassabis, D. (2021). Highly accurate protein structure prediction with alphafold. *Nature*, *596*(7873), 583–589. https://doi.org/10.1038/s41586-021-03819-2

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. Neural message passing for quantum chemistry (D. Precup & Y. W. Teh, Eds.). In: *Proceedings of the 34th international conference on machine learning* (D. Precup & Y. W. Teh, Eds.). Ed. by Precup, D., & Teh, Y. W. *70*. Proceedings of Machine Learning Research. PMLR, 2017, 1263–1272.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., & Bengio, Y. (2014b). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, *abs/1406.1078*.

Vinyals, O., Bengio, S., & Kudlur, M. (2016). Order matters: Sequence to sequence for sets.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, *9*(8), 1735â1780. https://doi.org/10.1162/neco.1997.9.8.1735

Dauparas, J., Anishchenko, I., Bennett, N., Bai, H., Ragotte, R. J., Milles, L. F., Wicky, B. I., Courbet, A., de Haas, R. J., Bethel, N., et al. (2022). Robust deep learning–based protein sequence design using proteinmpnn. *Science*, *378*(6615), 49–56.

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2023). Neural operator: Learning maps between function spaces with applications to pdes. *Journal of Machine Learning Research*, *24*(89), 1–97.

Satorras, V. G., Hoogeboom, E., & Welling, M. E(n) equivariant graph neural networks (M. Meila & T. Zhang, Eds.). In: *Proceedings of the 38th international conference on*

machine learning (M. Meila & T. Zhang, Eds.). Ed. by Meila, M., & Zhang, T. *139*. Proceedings of Machine Learning Research. PMLR, 2021, 9323–9332.

Kearnes, S., McCloskey, K., Berndl, M., Pande, V., & Riley, P. (2016). Molecular graph convolutions: Moving beyond fingerprints. *Journal of Computer-Aided Molecular Design*, *30*(8), 595–608. https://doi.org/10.1007/s10822-016-9938-8

Yang, K., Swanson, K., Jin, W., Coley, C., Eiden, P., Gao, H., Guzman-Perez, A., Hopper, T., Kelley, B., Mathea, M., et al. (2019). Analyzing learned molecular representations for property prediction. *Journal of chemical information and modeling*, *59*(8), 3370–3388.

Dai, H., Dai, B., & Song, L. Discriminative embeddings of latent variable models for structured data (M. F. Balcan & K. Q. Weinberger, Eds.). In: *Proceedings of the 33rd international conference on machine learning* (M. F. Balcan & K. Q. Weinberger, Eds.). Ed. by Balcan, M. F., & Weinberger, K. Q. *48*. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, 2702–2711.

Nair, V., & Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In: In *Proceedings of the 27th international conference on machine learning (icml-10)*. 2010, 807–814.

Maas, A. L., Hannun, A. Y., Ng, A. Y., et al. Rectifier nonlinearities improve neural network acoustic models. In: In *Proc. icml. 30*. (1). Atlanta, GA. 2013, 3.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition.

Chen, C., Ye, W., Zuo, Y., Zheng, C., & Ong, S. P. (2019). Graph networks as a universal machine learning framework for molecules and crystals. *Chemistry of Materials*, *31*(9), 3564â3572. https://doi.org/10.1021/acs.chemmater.9b01294

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gulcehre, C., Song, F., Ballard, A., Gilmer, J., Dahl, G., Vaswani, A., Allen, K., Nash, C., Langston, V., … Pascanu, R. (2018). Relational inductive biases, deep learning, and graph networks.

Zheng, H., Yang, Z., Liu, W., Liang, J., & Li, Y. Improving deep neural networks using softplus units. In: In *2015 international joint conference on neural networks (ijcnn)*. 2015, 1–4. https://doi.org/10.1109/IJCNN.2015.7280459

Xiong, Z., Wang, D., Liu, X., Zhong, F., Wan, X., Li, X., Li, Z., Luo, X., Chen, K., Jiang, H., & Zheng, M. (2020). Pushing the boundaries of molecular representation for drug discovery with the graph attention mechanism [PMID: 31408336]. *Journal of Medicinal Chemistry*, *63*(16), 8749–8760. https://doi.org/10.1021/acs.jmedchem.9b00959

Veličković, P., Cucurull, G., Casanova, A., Romero, A., LiÄ¿, P., & Bengio, Y. (2018). Graph attention networks. https://doi.org/1710.10903

Bahdanau, D., Cho, K., & Bengio, Y. (2016). Neural machine translation by jointly learning to align and translate.

Cho, K., van Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014a). On the properties of neural machine translation: Encoder-decoder approaches.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms

for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Fey, M., & Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In: In *Iclr workshop on representation learning on graphs and manifolds*. 2019.

Weininger, D. (1988). Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, *28*(1), 31–36. https://doi.org/10.1021/ci00057a005

Li, M., Zhou, J., Hu, J., Fan, W., Zhang, Y., Gu, Y., & Karypis, G. (2021). Dgl-lifesci: An open-source toolkit for deep learning on graphs in life science. *ACS Omega*.

Na, G. S., Kim, H. W., & Chang, H. (2020). Costless performance improvement in machine learning for graph-based molecular analysis [PMID: 31928003]. *Journal of Chemical Information and Modeling*, *60*(3), 1137–1145. https://doi.org/10.1021/acs.jcim.9b00816

Butina, D. (1999). Unsupervised data base clustering based on daylight's fingerprint and tanimoto similarity: A fast and automated way to cluster small and large data sets. *Journal of Chemical Information and Computer Sciences*, *39*(4), 747–750. https://doi.org/10.1021/ci9803381

Djoumbou Feunang, Y., Eisner, R., Knox, C., Chepelev, L., Hastings, J., Owen, G., Fahy, E., Steinbeck, C., Subramanian, S., Bolton, E., Greiner, R., & Wishart, D. S. (2016). Classyfire: Automated chemical classification with a comprehensive, computable taxonomy. *Journal of Cheminformatics*, *8*(1), 61. https://doi.org/10.1186/s13321-016-0174-y

Mobley, D. L., & Guthrie, J. P. (2014). FreeSolv: A database of experimental and calculated hydration free energies, with input files. *J. Comput. Aided Mol. Des.*, *28*(7), 711–720.

Bemis, G. W., & Murcko, M. A. (1996). The properties of known drugs. 1. molecular frameworks. *Journal of Medicinal Chemistry*, *39*(15), 2887–2893. https://doi.org/10.1021/jm9602928

Daylight theory manual [Accessed: 2024-03-04]. (2011).

Tanimoto, T. (1958). *An elementary mathematical theory of classification and prediction* (Internal Technical Report No. 1957). IBM.

Martin, E. J., Polyakov, V. R., Zhu, X.-W., Tian, L., Mukherjee, P., & Liu, X. (2019). All-assay-max2 pqsar: Activity predictions as accurate as four-concentration ic50s for 8558 novartis assays [PMID: 31518124]. *Journal of Chemical Information and Modeling*, *59*(10), 4450–4459. https://doi.org/10.1021/acs.jcim.9b00375

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In: In *Advances in neural information processing systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

Bradley, J.-C., Williams, A., & Lang, A. (2014). Jean-Claude Bradley Open Melting Point Dataset. https://doi.org/10.6084/m9.figshare.1031637.v2

Mansouri, K., Grulke, C. M., Richard, A. M., Judson, R. S., & Williams, A. J. (2016). An automated curation procedure for addressing chemical errors and inconsistencies in

public datasets used in qsar modelling. *SAR and QSAR in Environmental Research*, *27*(11), 939–965. https://doi.org/10.1080/1062936X.2016.1253611

Ulrich, N., Goss, K.-U., & Ebert, A. (2021). Exploring the octanol–water partition coefficient dataset using deep learning techniques and data augmentation. *Communications Chemistry*, *4*(1), 90. https://doi.org/10.1038/s42004-021-00528-9

Wu, Z., Ramsundar, B., Feinberg, E. N., Gomes, J., Geniesse, C., Pappu, A. S., Leswing, K., & Pande, V. S. (2017). Moleculenet: A benchmark for molecular machine learning. *CoRR*, *abs/1703.00564*.

Aouichaoui, A. R., Cogliati, A., Abildskov, J., & Sin, G. S-gnn: State-dependent graph neural networks for functional molecular properties (A. C. Kokossis, M. C. Georgiadis, & E. Pistikopoulos, Eds.). In: *33rd european symposium on computer aided process engineering* (A. C. Kokossis, M. C. Georgiadis, & E. Pistikopoulos, Eds.). Ed. by Kokossis, A. C., Georgiadis, M. C., & Pistikopoulos, E. Vol. 52. Computer Aided Chemical Engineering. Elsevier, 2023, pp. 575–581. https://doi.org/https://doi.org/10.1016/B978-0-443-15274-0.50091-3

Maćkiewicz, A., & Ratajczak, W. (1993). Principal components analysis (pca). *Computers Geosciences*, *19*(3), 303–342. https://doi.org/https://doi.org/10.1016/0098-3004(93)90090-R

van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, *9*(86), 2579–2605.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Detlefsen, N. S. (2024). Machine learning operations.

Aouichaoui, A. R., Fan, F., Abildskov, J., & Sin, G. (2023c). Application of interpretable group-embedded graph neural networks for pure compound properties. *Computers Chemical Engineering*, *176*, 108291. https://doi.org/https://doi.org/10.1016/j.compchemeng.2023.108291

What is density functional theory? (2009). In *Density functional theory* (pp. 1–33). John Wiley Sons, Ltd. https://doi.org/https://doi.org/10.1002/9780470447710.ch1

Aouichaoui, A. R. N., Fan, F., Mansouri, S. S., Abildskov, J., & Sin, G. (2023a). Combining group-contribution concept and graph neural networks toward interpretable molecular property models [PMID: 36716461]. *Journal of Chemical Information and Modeling*, *63*(3), 725–744. https://doi.org/10.1021/acs.jcim.2c01091

Kingma, D. P., & Ba, J. (2017). Adam: A method for stochastic optimization.

Falkner, S., Klein, A., & Hutter, F. (2018). Bohb: Robust and efficient hyperparameter optimization at scale.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & de Freitas, N. (2016). Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, *104*(1), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2018). Hyperband: A novel bandit-based approach to hyperparameter optimization.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., & Stoica, I. (2018). Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

Godbole, V., Dahl, G. E., Gilmer, J., Shallue, C. J., & Nado, Z. (2023). Deep learning tuning playbook [Version 1.0].

Murphy, K. P. (2023). *Probabilistic machine learning: Advanced topics*. MIT Press.

Win, Z.-M., Cheong, A. M. Y., & Hopkins, W. S. (2023). Using machine learning to predict partition coefficient (log p) and distribution coefficient (log d) with molecular descriptors and liquid chromatography retention time [PMID: 36926888]. *Journal of Chemical Information and Modeling*, *63*(7), 1906–1913. https://doi.org/10.1021/acs.jcim.2c01373

Zhou, G., Gao, Z., Ding, Q., Zheng, H., Xu, H., Wei, Z., Zhang, L., & Ke, G. Uni-mol: A universal 3d molecular representation learning framework. In: In *The eleventh international conference on learning representations*. 2023.

Lu, S., Gao, Z., He, D., Zhang, L., & Ke, G. (2023). Highly accurate quantum chemical property prediction with uni-mol+.

Heid, E., Greenman, K. P., Chung, Y., Li, S.-C., Graff, D. E., Vermeire, F. H., Wu, H., Green, W. H., & McGill, C. J. (2024). Chemprop: A machine learning package for chemical property prediction [PMID: 38147829]. *Journal of Chemical Information and Modeling*, *64*(1), 9–17. https://doi.org/10.1021/acs.jcim.3c01250

Zhang, Z., Guan, J., & Zhou, S. (2021). FraGAT: a fragment-oriented multi-scale graph attention model for molecular property prediction. *Bioinformatics*, *37*(18), 2981–2987. https://doi.org/10.1093/bioinformatics/btab195

Stärk, H., Beaini, D., Corso, G., Tossou, P., Dallago, C., Günnemann, S., & Lió, P. (2022). 3d infomax improves gnns for molecular property prediction.

Simeon, G., & de Fabritiis, G. (2023). Tensornet: Cartesian tensor representations for efficient learning of molecular potentials.

Gasteiger, J., Weißenberger, S., & Günnemann, S. (2019). Diffusion improves graph learning. *Advances in neural information processing systems*, *32*.

Chamberlain, B., Rowbottom, J., Gorinova, M. I., Bronstein, M., Webb, S., & Rossi, E. Grand: Graph neural diffusion (M. Meila & T. Zhang, Eds.). In: *Proceedings of the 38th international conference on machine learning* (M. Meila & T. Zhang, Eds.). Ed. by Meila, M., & Zhang, T. *139*. Proceedings of Machine Learning Research. PMLR, 2021, 1407–1418.

Li, Y., Wang, X., Liu, H., & Shi, C. (2024). A generalized neural diffusion framework on graphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, *38*(8), 8707–8715. https://doi.org/10.1609/aaai.v38i8.28716

Kipf, T. N., & Welling, M. (2016). Variational graph auto-encoders.

Liao, R., Li, Y., Song, Y., Wang, S., Nash, C., Hamilton, W. L., Duvenaud, D., Urtasun, R., & Zemel, R. S. (2020). Efficient graph generation with graph recurrent attention networks.

Jin, W., Barzilay, R., & Jaakkola, T. Junction tree variational autoencoder for molecular graph generation (J. Dy & A. Krause, Eds.). In: *Proceedings of the 35th international*

conference on machine learning (J. Dy & A. Krause, Eds.). Ed. by Dy, J., & Krause, A. *80*. Proceedings of Machine Learning Research. PMLR, 2018, 2323–2332.

Hukkerikar, A. S., Sarup, B., Ten Kate, A., Abildskov, J., Sin, G., & Gani, R. (2012). Group-contribution+ (gc+) based estimation of properties of pure components: Improved property estimation and uncertainty analysis. *Fluid Phase Equilibria*, *321*, 25–43. https://doi.org/https://doi.org/10.1016/j.fluid.2012.02.010

Appendix

A Butina Clustering

Steps of the Butina clustering algorithm (Butina, 1999):

1. Generate Daylight fingerprints ("Daylight Theory Manual", 2011): Fingerprints are a unique bit wise representation of the structural information of a molecule. This representation, purely consisting of 1 and 0's, is then ideal for comparative analysis.

2. Find the number of neighbors for every molecule. To do this, we find the Tanimoto similarity (Tanimoto, 1958) coefficient between each pair of molecule bitmaps. The coefficient is calculated as the total number of bits (structural information) in common and divided by the total number of bits. It ranges from 0 to 1 where 1 is the same molecule. Based on this number, we can find the number of neighbors corresponding to a molecule.

3. Clustering the data using exclusion spheres. Going from the molecule with the most similar neighbors, we assign molecules to it based on the Tanimoto similarity and a predefined threshold. Molecules assigned to a cluster cannot become centroids themselves.

The result is that a lot of very similar clusters are formed, leaving some remaining singletons that do not conform and would only create heterogenous clusters. The following is an algorithmic formulation of the above:

Algorithm 1 Butina clustering

Data: SMILES
Result: Cluster IDs
generate fingerprints
 for *all molecules* **do**
 | calculate Tanimoto similarity to all other molecules
end
threshold similarity to determine number of neighbors
 sort by number of neighbors from most to least
 $centroid \leftarrow$ first molecule of list
 while *molecules not assigned* **do**
 | **for** *molecules not assigned* **do**
 | | $target \leftarrow molecule$
 | | $similiarty \leftarrow$ tanimoto similarity$(centroid, target)$
 | | **if** $similarity < threshold$ **then**
 | | | assign molecule current cluster
 | | **end**
 | **end**
 | $centroid \leftarrow$ next molecule in sorted list
end

B Optimal Hyperparameters

Melting Point

Optimal value of:	MPNN	DMPNN	MEGNet	AFP
Initial learning rate	$9.56 \cdot 10^{-3}$	$4.65 \cdot 10^{-3}$	$1.00 \cdot 10^{-3}$	$1.22 \cdot 10^{-3}$
Weight decay	$9.43 \cdot 10^{-4}$	$8.66 \cdot 10^{-4}$	$1.00 \cdot 10^{-6}$	$1.22 \cdot 10^{-3}$
Hidden (node) dimensions	46	74	32	36
Node embedding layers (T)	1	1	1	1
Representation dropout	0.25	0.25	0.15	0.15
MLP layers	1	1	2	1
Graph embedding layers (L)	*N.A.*	*N.A.*	*N.A.*	1
Hidden edge dimension	*N.A.*	*N.A.*	128	*N.A.*

Table 6: The optimal hyperparameters for the Melting Point dataset. The original search spaces can be found in table 4.

Log-P

Optimal value of:	MPNN	DMPNN	MEGNet	AFP
Initial learning rate	$7.68 \cdot 10^{-3}$	$1.64 \cdot 10^{-2}$	$1.00 \cdot 10^{-3}$	$4.88 \cdot 10^{-3}$
Weight decay	$1.39 \cdot 10^{-4}$	$1.12 \cdot 10^{-3}$	$1.00 \cdot 10^{-6}$	$2.87 \cdot 10^{-4}$
Hidden (node) dimensions	69	94	128	67
Node embedding layers (T)	1	1	1	1
Representation dropout	0.13	0.00	0.00	0.12
MLP layers	1	1	4	1
Graph embedding layers (L)	*N.A.*	*N.A.*	*N.A.*	4
Hidden edge dimension	*N.A.*	*N.A.*	32	*N.A.*

Table 7: The optimal hyperparameters for the Log-P dataset. The original search spaces can be found in table 4.

Heat capacity

Optimal value of:	MPNN	DMPNN	MEGNet	AFP
Initial learning rate	$1.56 \cdot 10^{-3}$	$3.68 \cdot 10^{-3}$	$1.00 \cdot 10^{-3}$	$4.35 \cdot 10^{-3}$
Weight decay	$5.07 \cdot 10^{-4}$	$2.94 \cdot 10^{-3}$	$1.00 \cdot 10^{-6}$	$2.58 \cdot 10^{-4}$
Hidden (node) dimensions	156	180	32	34
Node embedding layers (T)	2	3	1	1
Representation dropout	0.34	0.34	0.00	0.17
MLP layers	4	1	2	1
Graph embedding layers (L)	*N.A.*	*N.A.*	*N.A.*	4
Hidden edge dimension	*N.A.*	*N.A.*	32	*N.A.*

Table 8: The optimal hyperparameters for the Heat capacity dataset. The original search spaces can be found in table 4.

FreeSolv

Optimal value of:	MPNN	DMPNN	MEGNet	AFP
Initial learning rate	$3.64 \cdot 10^{-2}$	$4.82 \cdot 10^{-3}$	$1.00 \cdot 10^{-3}$	$5.59 \cdot 10^{-3}$
Weight decay	$1.36 \cdot 10^{-3}$	$1.67 \cdot 10^{-3}$	$1.00 \cdot 10^{-6}$	$1.27 \cdot 10^{-3}$
Hidden (node) dimensions	65	100	32	126
Node embedding layers (T)	5	2	1	2
Representation dropout	0.05	0.35	0.00	0.01
MLP layers	2	1	2	1
Graph embedding layers (L)	*N.A.*	*N.A.*	*N.A.*	1
Hidden edge dimension	*N.A.*	*N.A.*	32	*N.A.*

Table 9: The optimal hyperparameters for the FreeSolv dataset. The original search spaces can be found in table 4.

C Weave Model Comparison

In this appendix, Weave is briefly examined and why it was chosen to be excluded from the main experiments. Weave, like mentioned in section 5.3, does not leverage any powerful architecture like the other models and therefore falls behind in performance. After using the same search-space and optimization scheme discussed in section 5.4, specifically the BOHB algorithm, the model is evaluated and the results presented in table 10.

Split	MAE	RSME	MDAPE	R^2
Test	61.42	79.01	66.03	0.27
Overall	62.43	79.45	67.24	0.28

Table 10: All the evaluation metrics of the Weave model trained and tested on the melting point dataset. The same metrics for the main text models can be found in table 5.

It is obvious that Weave performs significantly worse than the other models implemented in the toolbox, and should only be used as a baseline for property prediction.

The ideas behind the model should not be disregarded, however. Learning the node and edge embedding at the same time can certainly be useful in the right context. For instance, if the objective is to generate molecules from scratch, Weave might have an advantage over the other models as it learns to predict both atom and bond features.

D Consistency Checking Models with Melting Point Extrapolation

This appendix serves as a demonstration of how one might perform a consistency check on a chemical property prediction model like the ones implemented in `GraPE-Chem`. In machine learning, it is often not easy to test the predictive quality of a model without acquiring new data, as not all models are concerned with properties that can be checked visually. Instead, one could try to guarantee that a model obeys physical laws that have intuitive properties.

In this context, the melting point of long carbon chains is used as such a physical law. The intuition is then, that the melting point continues to increase for every new member of the chain. To check if this applies, the four models from the main text are loaded and used to predict the melting points of carbon chains consisting of n carbons ($[C]$). The reference model is Group Contribution + by Hukkerikar et al., 2012, a very reliable property estimation model. The results can be found in figure 17.

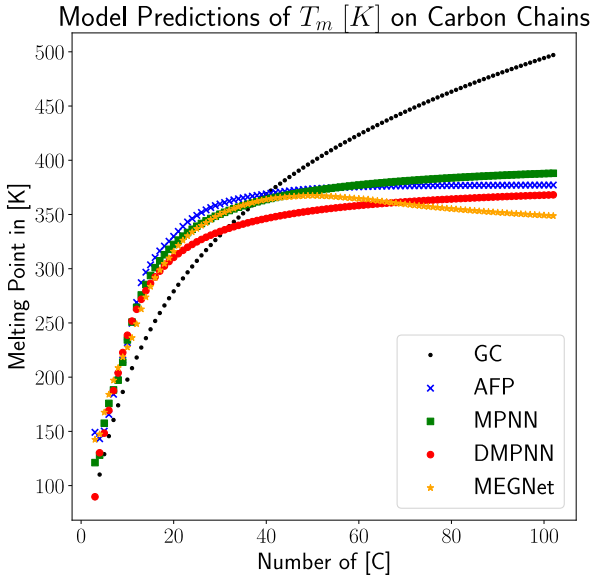


Figure 17: The melting point predictions on long carbon chains for all four main models discussed in the main text as well as the Group Contribution model by Hukkerikar et al., 2012.

Initially, all five model predictions increase exponentially until about 20 carbons per chain, where the four models from the main text taper off while the GC model continues to increase. While the exact behaviour of the T_m for such long chains is not obvious, the temperature is still expected to increase. For this reason, the falling melting point predictions of the MEGNet model (in yellow) is very suspicious and likely indicates that the model was not trained correctly (possibly due to over or underfitting).

E General Demonstration of GraPE-Chem

GraPE-Chem Demonstration

June 10, 2024

1 General Demo

This notebook serves as a demonstration of what the toolbox can do. This project is a work in progress, so there will be updates to the code and subsequently the applications. Below are some highlights of what is currently possible.

1.1 Datasets

The starting point of the project is the ability to easily load chemical datasets into a format that is usable from a Machine Learning perspective. All the relevant datasets make use of **SMILES** (Simplified molecular-input line-entry system) to store information on molecules. The in-built `DataSet` objects then take the SMILES representation of molecules together with a target array and any other additional information, and load then into **graphs**.

Below is an example of an implemented dataset that can, similarly to the torch datasets, download and load a dataset from the internet.

1.1.1 Jean-Claude Bradley Open Melting Point Dataset [1]

```
from grape_chem.datasets import BradleyDoublePlus
print(f'length of dataset at the start: {3024}\n-------------')
data = BradleyDoublePlus(log=True, only_organic=True)
print(f'-------------\nlength of dataset at the end: {len(data)}')
```

```
length of dataset at the start: 3024
-------------
SMILES [O-][N+]#N in index 1 does not contain at least one carbon and will be
ignored.
SMILES FS(F)(=O)=O in index 2 does not contain at least one carbon and will be
ignored.
SMILES BrBr in index 4 does not contain at least one carbon and will be ignored.
SMILES II in index 8 does not contain at least one carbon and will be ignored.
SMILES ClS(Cl)(=O)=O in index 79 does not contain at least one carbon and will
be ignored.
SMILES B(OCC)(OCC)OCC in index 144 contains the atom B that is not permitted and
will be ignored.
SMILES c1cccn1 in index 308 is not valid.
SMILES N#Cc3cncc3c1cccc2OC(F)(F)Oc12 in index 370 is not valid.
SMILES CC[Si](CC)(CC)CC in index 519 contains the atom Si that is not permitted
```

```
and will be ignored.
SMILES CC[Si](CC)(Cl)Cl in index 541 contains the atom Si that is not permitted
and will be ignored.
SMILES [SiH](Cl)(Cl)Cl in index 565 contains the atom Si that is not permitted
and will be ignored.
SMILES [SiH](Cl)(Cl)Cl in index 565 does not contain at least one carbon and
will be ignored.
SMILES c1ccc(cc1)[Si](c2ccccc2)(c3ccccc3)c4ccccc4 in index 592 contains the atom
Si that is not permitted and will be ignored.
SMILES c1ccc(cc1)[Si]2(O[Si](O[Si](O[Si](O2)(c3ccccc3)c4ccccc4)(c5ccccc5)c6ccccc
6)(c7ccccc7)c8ccccc8)c9ccccc9 in index 605 contains the atom Si that is not
permitted and will be ignored.
SMILES c1ccc(cc1)[Si]2(O[Si](O[Si](O[Si](O2)(c3ccccc3)c4ccccc4)(c5ccccc5)c6ccccc
6)(c7ccccc7)c8ccccc8)c9ccccc9 in index 605 contains the atom Si that is not
permitted and will be ignored.
SMILES c1ccc(cc1)[Si]2(O[Si](O[Si](O[Si](O2)(c3ccccc3)c4ccccc4)(c5ccccc5)c6ccccc
6)(c7ccccc7)c8ccccc8)c9ccccc9 in index 605 contains the atom Si that is not
permitted and will be ignored.
SMILES c1ccc(cc1)[Si]2(O[Si](O[Si](O[Si](O2)(c3ccccc3)c4ccccc4)(c5ccccc5)c6ccccc
6)(c7ccccc7)c8ccccc8)c9ccccc9 in index 605 contains the atom Si that is not
permitted and will be ignored.
SMILES C[Si]1(O[Si](O[Si](O[Si](O1)(C)C=C)(C)C=C)(C)C=C)C=C in index 626
contains the atom Si that is not permitted and will be ignored.
SMILES C[Si]1(O[Si](O[Si](O[Si](O1)(C)C=C)(C)C=C)(C)C=C)C=C in index 626
contains the atom Si that is not permitted and will be ignored.
SMILES C[Si]1(O[Si](O[Si](O[Si](O1)(C)C=C)(C)C=C)(C)C=C)C=C in index 626
contains the atom Si that is not permitted and will be ignored.
SMILES C[Si]1(O[Si](O[Si](O[Si](O1)(C)C=C)(C)C=C)(C)C=C)C=C in index 626
contains the atom Si that is not permitted and will be ignored.
SMILES COc1cc(cc(OC)c1OC)/C=C/C(=O)O[C@@H]3C[C@@H]4CN5CCc2c6ccc(OC)cc6nc2[C@H]5C
[C@@H]4[C@@H]([C@H]3OC)C(=O)OC in index 662 is not valid.
SMILES c1cncn1 in index 697 is not valid.
SMILES c1cccc2nccc12 in index 698 is not valid.
SMILES Cc1cc2nnnc2cc1 in index 728 is not valid.
SMILES C[Si]1(C)O[Si](C)(C)O[Si](C)(C)O[Si](C)(C)O1 in index 734 contains the
atom Si that is not permitted and will be ignored.
SMILES C[Si]1(C)O[Si](C)(C)O[Si](C)(C)O[Si](C)(C)O1 in index 734 contains the
atom Si that is not permitted and will be ignored.
SMILES C[Si]1(C)O[Si](C)(C)O[Si](C)(C)O[Si](C)(C)O1 in index 734 contains the
atom Si that is not permitted and will be ignored.
SMILES C[Si]1(C)O[Si](C)(C)O[Si](C)(C)O[Si](C)(C)O1 in index 734 contains the
atom Si that is not permitted and will be ignored.
SMILES C in index 819 consists of less than 2 heavy atoms and will be ignored.
SMILES c1ccc(cc1)[SiH](c2ccccc2)c3ccccc3 in index 842 contains the atom Si that
is not permitted and will be ignored.
SMILES Br in index 903 consists of less than 2 heavy atoms and will be ignored.
SMILES O=S=O in index 905 does not contain at least one carbon and will be
ignored.
```

SMILES O=C(OC)[C@@]2(C[C@H]4C[C@@](O)(CC)C[N@@](CCc1c3ccccc3nc12)C4)c5cc9c(cc5OC)N(C=O)[C@@H]6[C@]98CCN7CC=C[C@@](CC)([C@@H](OC(C)=O)[C@]6(O)C(=O)OC)[C@H]78 in index 933 is not valid.
SMILES C[Si](C)(C)C in index 935 contains the atom Si that is not permitted and will be ignored.
SMILES [O-][N+](=O)c1ccc2ncnc2c1 in index 1305 is not valid.
SMILES [O-][N+](=O)c1cnc(C)n1 in index 1364 is not valid.
SMILES ClS(Cl)=O in index 1384 does not contain at least one carbon and will be ignored.
SMILES C[Si]1(N[Si](N[Si](N[Si](N1)(C)C)(C)C)(C)C)C in index 1732 contains the atom Si that is not permitted and will be ignored.
SMILES C[Si]1(N[Si](N[Si](N[Si](N1)(C)C)(C)C)(C)C)C in index 1732 contains the atom Si that is not permitted and will be ignored.
SMILES C[Si]1(N[Si](N[Si](N[Si](N1)(C)C)(C)C)(C)C)C in index 1732 contains the atom Si that is not permitted and will be ignored.
SMILES C[Si]1(N[Si](N[Si](N[Si](N1)(C)C)(C)C)(C)C)C in index 1732 contains the atom Si that is not permitted and will be ignored.
SMILES C[Si](C)(C)[Si](C)(C)C in index 1750 contains the atom Si that is not permitted and will be ignored.
SMILES C[Si](C)(C)[Si](C)(C)C in index 1750 contains the atom Si that is not permitted and will be ignored.
SMILES c1cccc2ncnc12 in index 1820 is not valid.
SMILES B(c1ccccc1)(O)O in index 1983 contains the atom B that is not permitted and will be ignored.
SMILES CC[SiH2]CC in index 2256 contains the atom Si that is not permitted and will be ignored.
SMILES c1ccc(cc1)[Si](c2ccccc2)(c3ccccc3)Cl in index 2268 contains the atom Si that is not permitted and will be ignored.
SMILES CC(=O)O[Si](C)(OC(=O)C)OC(=O)C in index 2316 contains the atom Si that is not permitted and will be ignored.
SMILES C[Si](c1ccccc1)(c2ccccc2)c3ccccc3 in index 2325 contains the atom Si that is not permitted and will be ignored.
SMILES Cc2cnc1ccccc12 in index 2335 is not valid.
SMILES c1ccc(cc1)[Si](c2ccccc2)(c3ccccc3)O in index 2390 contains the atom Si that is not permitted and will be ignored.
SMILES c1cccc2nnnc12 in index 2516 is not valid.
SMILES B(c1ccccc1C)(O)O in index 2674 contains the atom B that is not permitted and will be ignored.
SMILES CCO[Si](OCC)(OCC)OCC in index 2681 contains the atom Si that is not permitted and will be ignored.
SMILES n2cnc(NCc1ccccc1)c3ncnc23 in index 2686 is not valid.
SMILES B(c1ccccc1)(c2ccccc2)OCCN in index 2741 contains the atom B that is not permitted and will be ignored.
SMILES C[Si](C)(C)C#C[Si](C)(C)C in index 2775 contains the atom Si that is not permitted and will be ignored.
SMILES C[Si](C)(C)C#C[Si](C)(C)C in index 2775 contains the atom Si that is not permitted and will be ignored.
SMILES n1c3ccccc3nc1c2cscn2 in index 2916 is not valid.

```
SMILES COB(OC)OC in index 2935 contains the atom B that is not permitted and
will be ignored.
SMILES N[C@@H](Cc1cncn1)C(=O)O in index 2944 is not valid.
SMILES C[Si]1(C)O[Si](C)(C)O[Si](C)(C)O1 in index 3008 contains the atom Si that
is not permitted and will be ignored.
SMILES C[Si]1(C)O[Si](C)(C)O[Si](C)(C)O1 in index 3008 contains the atom Si that
is not permitted and will be ignored.
SMILES C[Si]1(C)O[Si](C)(C)O[Si](C)(C)O1 in index 3008 contains the atom Si that
is not permitted and will be ignored.
SMILES Sc1nc2ccccc2n1 in index 3018 is not valid.
-------------
length of dataset at the end: 2989
```

As seen above, the data loader filters the SMILES based a few different criteria. One of them is a check if rdkit recognizes the SMILES representation as valid, another one is whether the SMILES atoms are **all** part of the *allowed* atom symbols. After filtering, we can access the information using the regular class notations.

1.1.2 Example outputs

We can access the SMILES from the melting point dataset as such:

```
# SMILES
data.smiles[0:5]
```

```
array(['CC1CCC1', 'CC(C)N(CCC(C(N)=O)(c1ccccc1)c1ccccn1)C(C)C',
       'CCn1cc(C(=O)O)c(=O)c2cc(F)c(N3CCNC(C)C3)c(F)c21', 'CN(C)C',
       'ClC(Cl)(Cl)Cl'], dtype=object)
```

And their corresponding graphs are stored at the same indices.

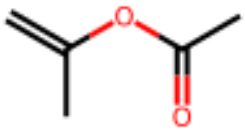
```
# Their corresponding graphs, saved using the torch geometric Data objects.
data[0:5]
```

```
[Data(x=[5, 44], edge_index=[2, 10], edge_attr=[10, 12], y=[1]),
 Data(x=[25, 44], edge_index=[2, 52], edge_attr=[52, 12], y=[1]),
 Data(x=[25, 44], edge_index=[2, 54], edge_attr=[54, 12], y=[1]),
 Data(x=[4, 44], edge_index=[2, 6], edge_attr=[6, 12], y=[1]),
 Data(x=[5, 44], edge_index=[2, 8], edge_attr=[8, 12], y=[1])]
```

We can draw any of the loaded SMILES using rdkit directly:

```
print('SMILES:', data.smiles[10])
print('Target: ', data.target[10], ' [Celsius]')
print('Graph Target: ', data.graphs[10].y, ' [Celsius]')
data.draw_smile(10)
```

```
SMILES: C=C(C)OC(C)=O
Target:  -93.0  [Celsius]
Graph Target:  tensor([-93.])  [Celsius]
```

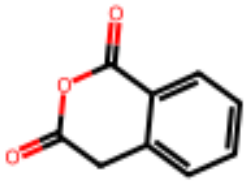



```
print('SMILES:', data.smiles[100])
print('Target: ', data.target[100], ' [Celsius]')
print('Graph Target: ', data.graphs[100].y, ' [Celsius]')
data.draw_smile(100)
```

```
SMILES: CCCCCCCCCCCCCCCCCCBr
Target:  28.0  [Celsius]
Graph Target:  tensor([28.])  [Celsius]
```

```
print('SMILES:', data.smiles[1000])
print('Target: ', data.target[1000], ' [Celsius]')
print('Graph Target: ', data.graphs[1000].y, ' [Celsius]')
data.draw_smile(1000)
```

```
SMILES: O=C1Cc2ccccc2C(=O)O1
Target:  142.0  [Celsius]
Graph Target:  tensor([142.])  [Celsius]
```



We can also save and load the dataset as such:

```
# We can also save and load the dataset as such:
# Datasets are saved using pickle, which allows for fast saving and loading.␣
 ↪Saving a dataset and then loading instead of loading from, for example, an␣
 ↪excel file is about 10 to 20 times faster.

from grape_chem.utils import DataSet

data.save_dataset('BradleyDoublePlus')
loaded_dataset = DataSet(file_path='./data/processed/BradleyDoublePlus.pickle')
loaded_dataset.smiles[0:5]
```

```
File saved at: ./data/processed/BradleyDoublePlus.pickle
Loaded dataset.
```

```
array(['CC1CCC1', 'CC(C)N(CCC(C(N)=O)(c1ccccc1)c1ccccn1)C(C)C',
       'CCn1cc(C(=O)O)c(=O)c2cc(F)c(N3CCNC(C)C3)c(F)c21', 'CN(C)C',
       'ClC(Cl)(Cl)Cl'], dtype=object)
```

Datasets are saved using pickle, which allows for fast saving and loading. Saving a dataset and then loading from the pickle directly instead of regenerating it from the orignal data is about 10 to 20 times faster.

1.2 Analysis

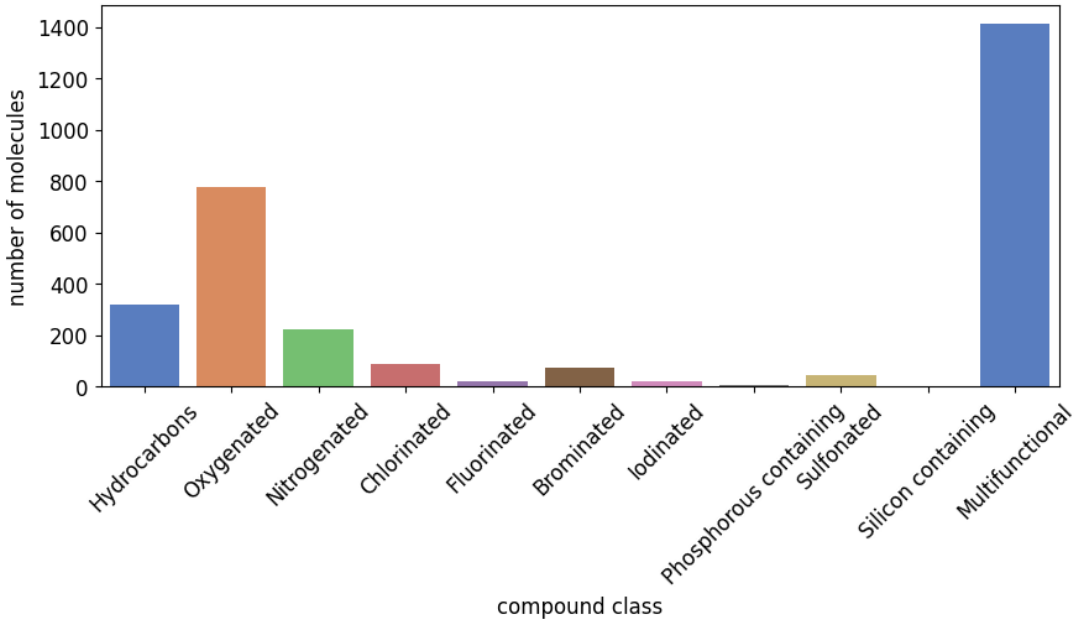
There are several options to analyze a loaded dataset. Below are some of these options.

1.2.1 Naive clustering

This chart is generated using a simple clustering algorithm that checks the letter in the SMILES and puts them into the below seen molecule classes. For example, if a SMILES only contains an 'O' and no other class letter, then it is part of 'Oxygenated'. If it contains 'O' and 'Cl', then it is part of 'Multifunctional'.

```
from grape_chem.plots import compound_nums_chart

compound_nums_chart(data.smiles, fig_size=(10,4))
```

```
<Axes: xlabel='compound class', ylabel='number of molecules'>
```



1.2.2 Classyfire

An almost always more useful clustering is using the Classyfire [2] classification done by Feunang et al.. In code terms, we pull the online documentation of the molecules in question with their Classyfire information, read it and cluster them based on that. This approach is far superior over the simple one above, but can take very long to do (it takes about 2 min to retrieve the information for 100 molecules).

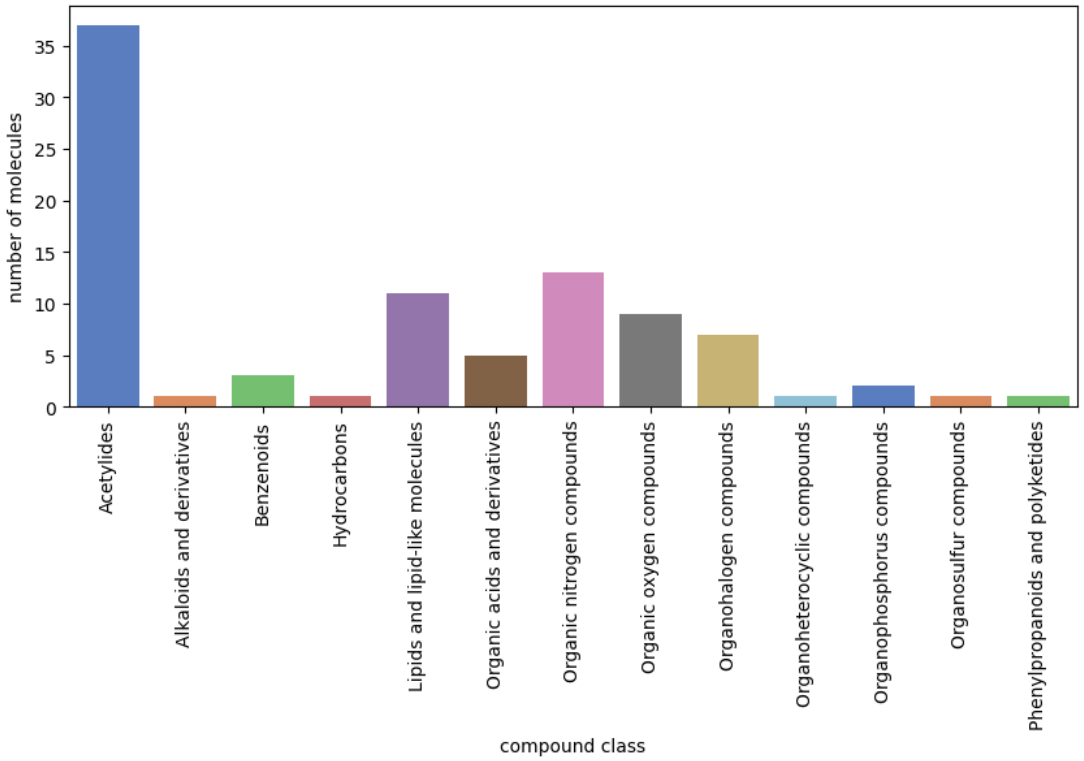
```
subset_smiles = data.smiles[100:200]

from grape_chem.analysis import classyfire, classyfire_result_analysis
ids, data_ids = classyfire(subset_smiles, log=False)
smile_classes, class_num_dictionary = classyfire_result_analysis(idx=ids)
print(class_num_dictionary)
```

```
Found log file in working directory.
All passed smiles are already in the passed log_file.
Key error occurred using superclass for file 683.json.
Key error occurred using superclass for file 695.json.
Key error occurred using superclass for file 656.json.
Key error occurred using superclass for file 640.json.
Key error occurred using superclass for file 660.json.
Key error occurred using superclass for file 621.json.
Key error occurred using superclass for file 699.json.
Key error occurred using superclass for file 676.json.
{'Benzenoids': 37, 'Phenylpropanoids and polyketides': 1, 'Organic nitrogen
compounds': 3, 'Organophosphorus compounds': 1, 'Organoheterocyclic compounds':
```

```
11, 'Organic oxygen compounds': 5, 'Organohalogen compounds': 13,
'Hydrocarbons': 9, 'Organic acids and derivatives': 7, 'Acetylides': 1, 'Lipids
and lipid-like molecules': 2, 'Alkaloids and derivatives': 1, 'Organosulfur
compounds': 1}
```

```
from grape_chem.plots import num_chart
num_chart(class_num_dictionary, fig_size=(10,4))
```

```
(<Figure size 1000x400 with 1 Axes>,
 <Axes: xlabel='compound class', ylabel='number of molecules'>)
```



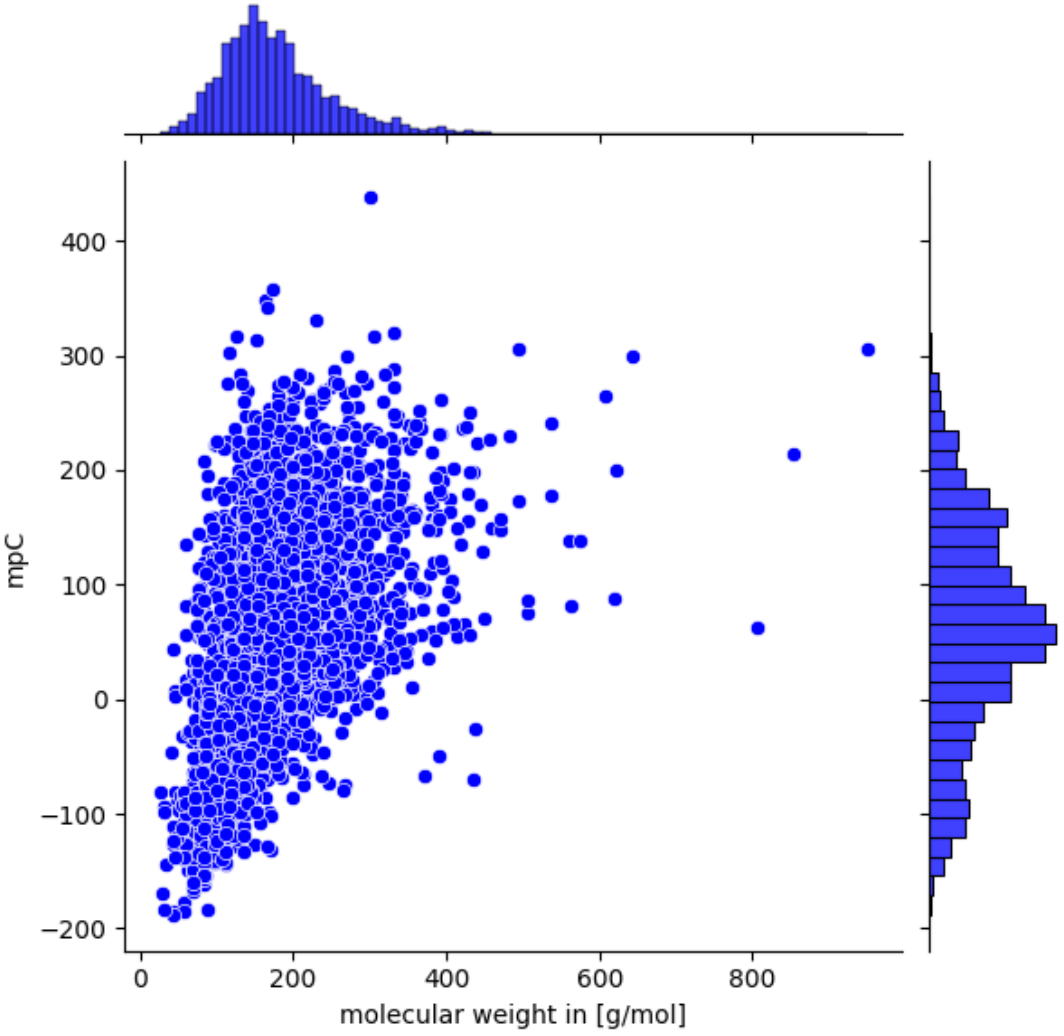
The classyfire code is split in two: (1) `classyfire` sends the relevant information to the classyfire website (http://classyfire.wishartlab.com/) and retrieves the class information as a json file. Furthermore, a csv file called `recorded_SMILES` is generated that stores the json file indices together with corresponding SMILES to prevent double downloading. (2) `classyfire_result_analysis` reads the json files and return dictionaries with all SMILES and their class as well as class frequency dictionary. That dictionary can then be fed directly to a class frequency plotting function.

1.2.3 Molecular Weight against Target

Another option for visual analysis is to plot the molecule weight against the target attribute. This might give an indication on how molecule size correlates with the target, often a usual observation.

```
from grape_chem.plots import mol_weight_vs_target
print(data.target)
mol_weight_vs_target(data.smiles, data.target, save_fig=True, fig_height=6,␣
↪target_name='mpC')
```

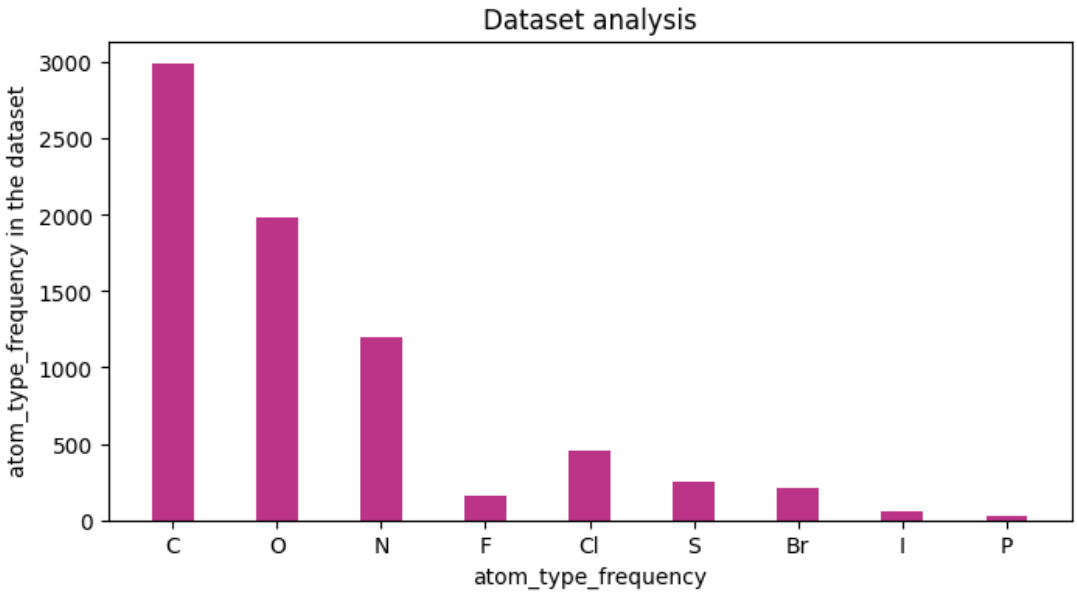
```
[-161.51   94.8   239.75 …  176.     65.    -34.  ]
```

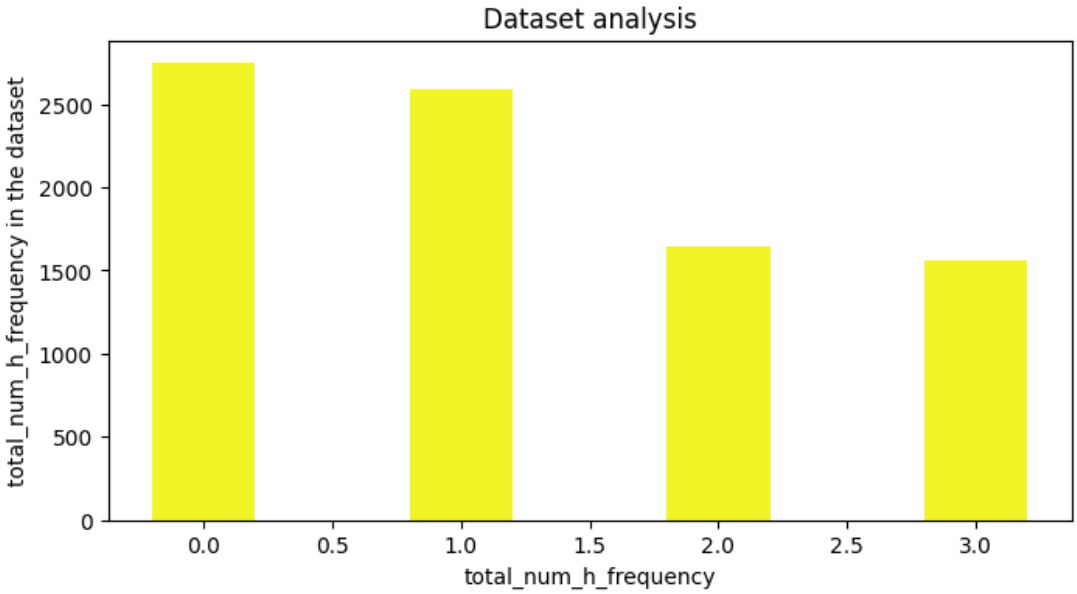


1.2.4 Feature number charts

If only a basic analysis of the dataset, is needed, then one can generate number charts based on the featurizion of the molecules. The implementation is built on top of DGL-LIFESCI `analyze_mols` function (github), below is an example.

```
%matplotlib inline

results, figures = data.analysis(download=True,␣
 ↪plots=['atom_type_frequency','total_num_h_frequency'], fig_size=[8,4],
                                  save_plots=True)
```

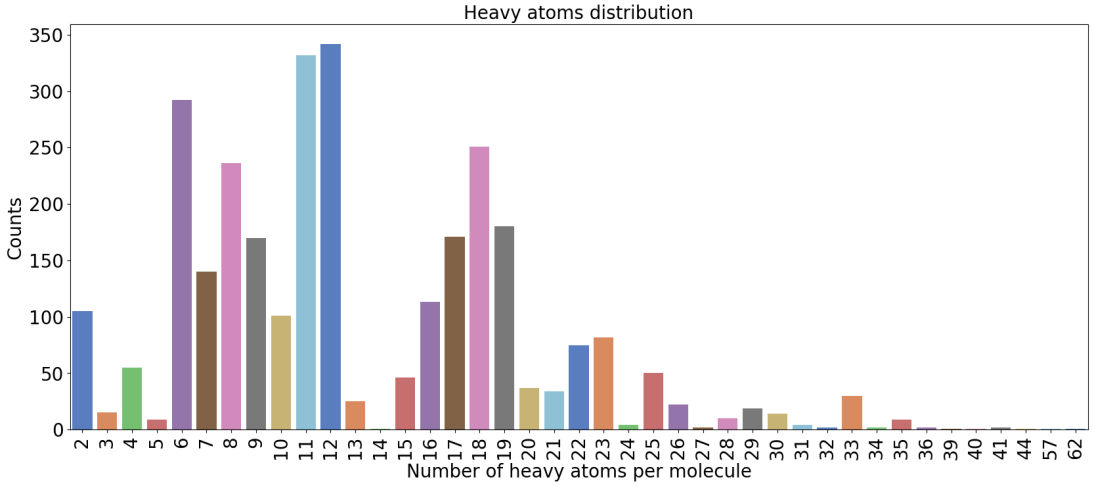




We could also plot the number of heavy atoms as a barchart:

```
from grape_chem.plots import num_heavy_plot
num_heavy_plot(data.smiles, fig_size=(20,8))
```

```
<Axes: title={'center': 'Heavy atoms distribution'}, xlabel='Number of heavy
atoms per molecule', ylabel='Counts'>
```



1.3 Clustering and data splitting

There are several ways to split the graph dataset into the training, validation and testing splits, including random, stratified or by molecular weight. A more interesting addition in this toolbox is a split based on **Butina clustering** [2], where the molecules are clustered based on the Morgan Fingerprint and the Tanimoto similarity. Within there are two options: (1) a **uniform** split where we sample across all clusters evenly, and (2) a **realistic** split where we fill up the train., val. and test. split with clusters from largest to smallest in order.

Below, we choose the *realistic* split:

```
from grape_chem.datasets import BradleyDoublePlus
data = BradleyDoublePlus()

train, val, test = data.split_and_scale(scale=True,␣
 ↪split_type='butina_realistic', seed=42, is_dmpnn=True);
```

```
100%|      | 2402/2402 [00:00<00:00, 4044.18it/s]
100%|      | 292/292 [00:00<00:00, 3589.26it/s]
100%|      | 295/295 [00:00<00:00, 3060.39it/s]
```

Note that we scale using the training set immediately and that we specify that we are working with DMPNN (a directional GNN).

1.4 GNN Models

With the data loaded, filtered, featurized and split, let's define a Graph Neural Network and test it! We can always find the number of features like thus:

```
print(f'Node feature dimension: {data.num_node_features}')
print(f'Edge feature dimension: {data.num_edge_features}')
```

```
Node feature dimension: 44
Edge feature dimension: 12
```

1.4.1 Model definition

To define a model, all we need to do is load one of the in-built models like `MPNN` or `MEGNet` and apply it. Any model that can handle the PyG graphs as input would work for that matter, fx. one could use the PyG models instead. Below, we load the `DMPNN` model directly from the package and initialize it:

```
from grape_chem.models import DMPNN
import torch

node_hidden_dim = 64
batch_size = 32
mlp_layers = [512, 256, 128]


model = DMPNN(node_in_dim=data.num_node_features, edge_in_dim=data.
 ↪num_edge_features, node_hidden_dim=node_hidden_dim,
              mlp_out_hidden=mlp_layers)



print('Full model:\n-------------------------------------------------')
print(model)

device = torch.device('cpu')
```

```
Full model:
-------------------------------------------------
DMPNN(
  (rep_dropout): Dropout(p=0.0, inplace=False)
  (encoder): DMPNNEncoder(
    (act_func): ReLU()
    (W1): Linear(in_features=56, out_features=64, bias=True)
    (W2): Linear(in_features=64, out_features=64, bias=True)
    (W3): Linear(in_features=108, out_features=64, bias=True)
    (dropout_layer): Dropout(p=0.15, inplace=False)
  )
  (mlp_out): Sequential(
```

```
    (0): Linear(in_features=64, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=128, bias=True)
    (5): ReLU()
    (6): Linear(in_features=128, out_features=1, bias=True)
  )
)
```

1.4.2 Loss and Optimizer

Like with any deep learning model, we define a loss function and optimizer.

```
from torch import nn

loss_func = nn.functional.mse_loss
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-6)
```

We can additionally define an Early Stopper to help improve the output:

```
from grape_chem.utils import EarlyStopping
early_stopper = EarlyStopping(patience=30, model_name='best_model')
```

As well as a scheduler to reduce the learning-rate whenever the training hits a plateau:

```
from torch.optim import lr_scheduler
scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.9,␣
 ↪min_lr=0.0000000000001, patience=15)
```

1.4.3 Training

Here, we just use the previous determined training and validation splits to train the model inside the `train_model` function:

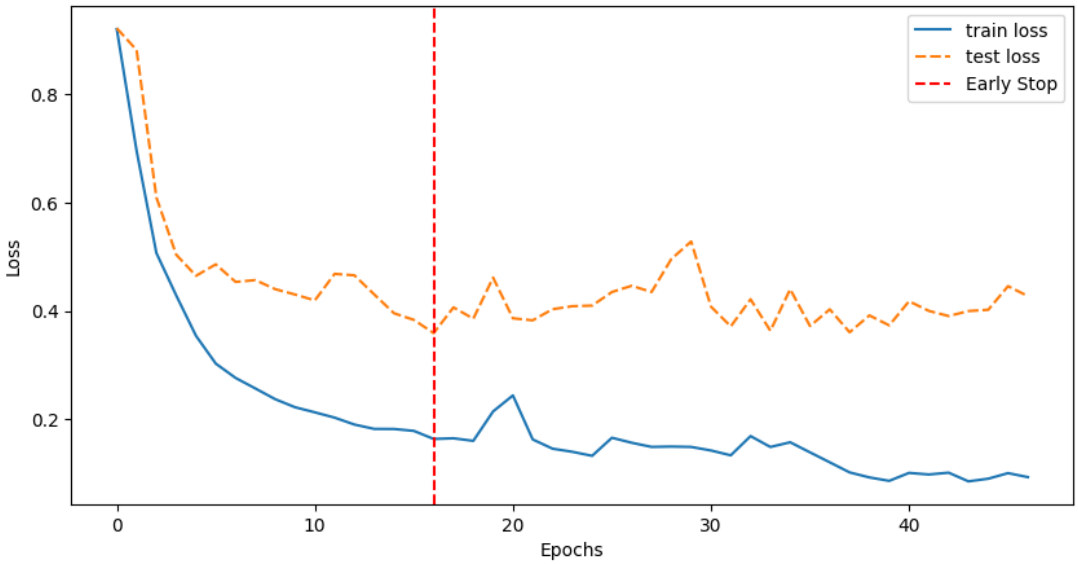
```
from grape_chem.utils import train_model

train_loss, val_loss = train_model(model = model,
                                   loss_func = 'mse',
                                   optimizer = optimizer,
                                   train_data_loader= train,
                                   val_data_loader = val,
                                   batch_size=batch_size,
                                   epochs=300,
                                   early_stopper=early_stopper,
                                    scheduler=scheduler)

model.load_state_dict(torch.load('best_model.pt'))
```

```
epoch=46, training loss= 0.093, validation loss= 0.427:  15%|          | 46/300
[00:14<01:20,  3.16it/s]

Early stopping reached with best validation loss 0.3590
Model saved at: best_model.pt


 <All keys matched successfully>
```

1.4.4 Loss plot

The training function returns the training and validation losses which can visualize as:

```
from grape_chem.plots import loss_plot
loss_plot([train_loss, val_loss], ['train loss', 'test loss'], early_stopper.
 ↪stop_epoch)
```



1.5 Testing and Post-processing

For testing, we use the `test_model` from the toolbox which essentially just uses predicts the test SMILES using the trained model:

```
from grape_chem.utils import test_model
preds = test_model(model=model,
                   test_data_loader=test)
```

```
100%|      | 10/10 [00:00<00:00, 178.52it/s]
```

Then we calculate some (re-scaled) metrics on the predictions using `pred_metric`:

```
from grape_chem.utils import pred_metric
pred_metric(prediction=preds,target=test.y, metrics='all', print_out=True,␣
  ↪rescale_data=data);
```

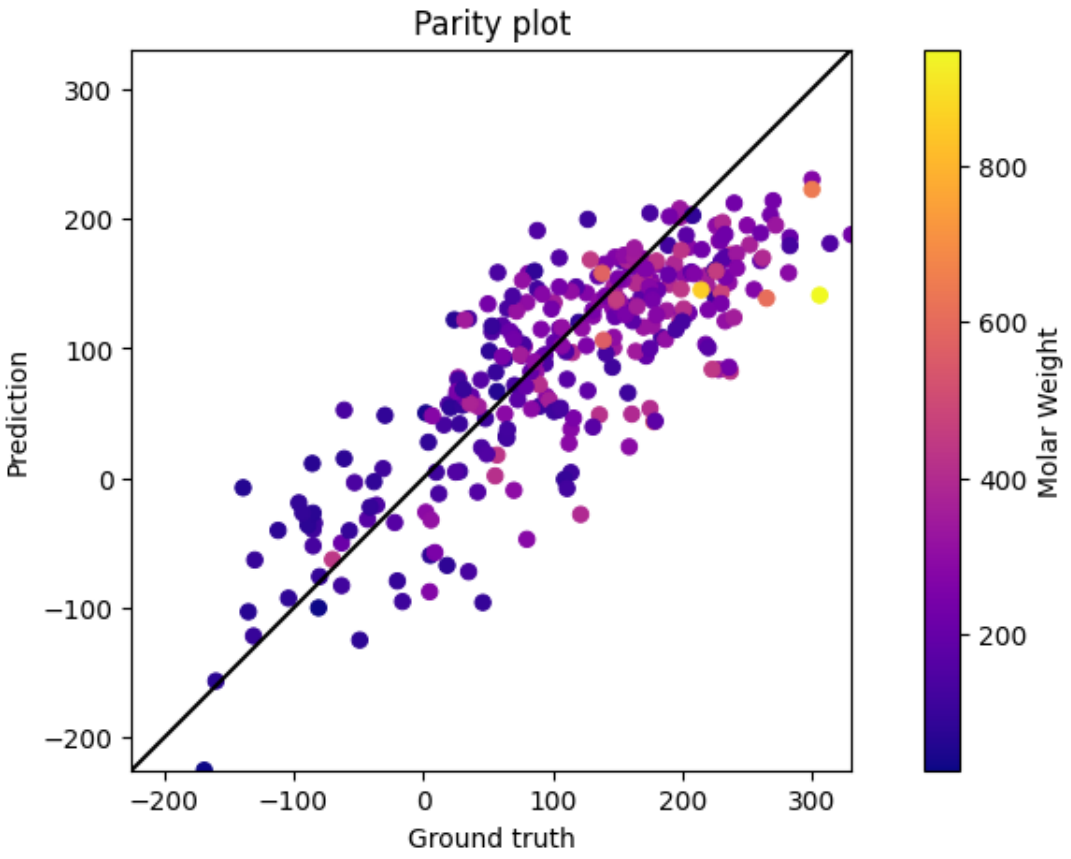
```
MSE: 3788.606
RMSE: 61.552
SSE: 1117638.819
MAE: 49.257
R2: 0.629
MRE: 64.075%
MDAPE: 28.588%
```

Note that we have to pass the original data object to make sure we are recaling on the correct data.

1.5.1 Parity and Residual plots

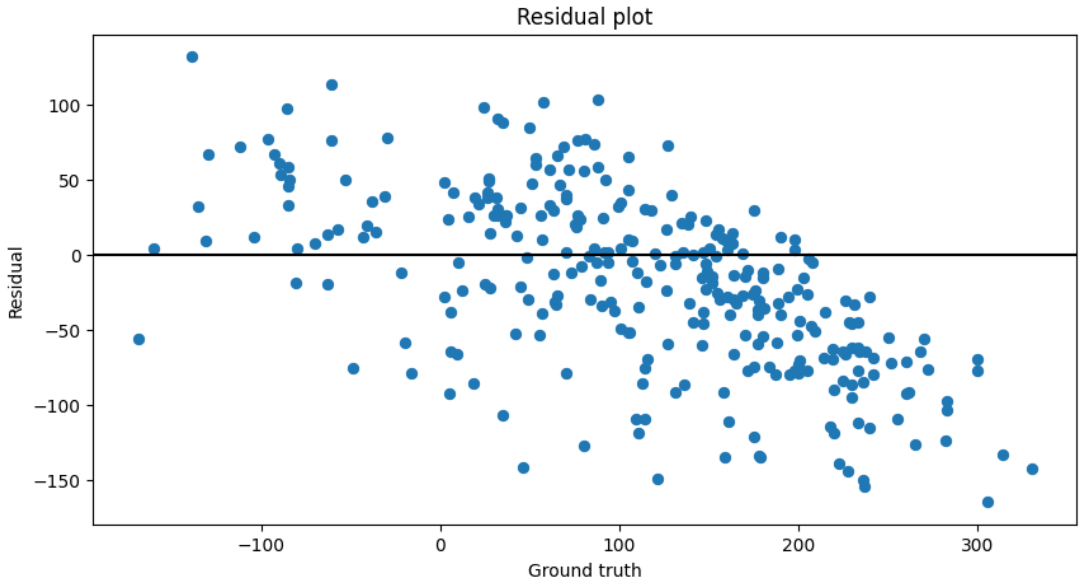
We could also choose to plot the parity and residual plots:

```
from grape_chem.plots import parity_plot, residual_plot
# We rescale the predictions and ground truth before plotting
preds_rescaled, test_y_rescaled = data.rescale_data(preds), data.
  ↪rescale_data(test.y)
parity_plot(prediction=preds_rescaled,target=test_y_rescaled, mol_weights=test.
  ↪mol_weights);
```




```
residual_plot(prediction=preds_rescaled,target=test_y_rescaled)
```

```
<Axes: title={'center': 'Residual plot'}, xlabel='Ground truth',
ylabel='Residual'>
```



1.5.2 Metrics

We can also examine the how the over splits, training and validation, behave. To do so, we might calculate average or **overall** metrics:

```
# Generating the predictions

train_preds = test_model(model=model,test_data_loader=train)
val_preds = test_model(model=model,test_data_loader=val)
test_preds = test_model(model=model,test_data_loader=test)
```

```
100%|      | 76/76 [00:00<00:00, 442.06it/s]
100%|      | 10/10 [00:00<00:00, 370.22it/s]
100%|      | 10/10 [00:00<00:00, 448.13it/s]
```

```
# Overall R2

overall = 0
overall += pred_metric(prediction=train_preds,target=train.y, metrics='r2',␣
 ↪print_out=False)['r2']
overall += pred_metric(prediction=val_preds,target=val.y, metrics='r2',␣
 ↪print_out=False)['r2']
overall += pred_metric(prediction=test_preds,target=test.y, metrics='r2',␣
 ↪print_out=False)['r2']
print(f'Overall R2: {overall/3}')
```

```
Overall R2: 0.7306092634540665
```

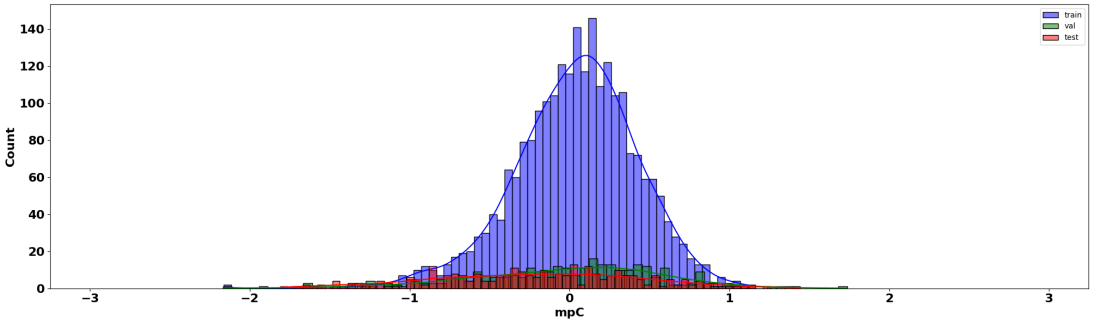
```
# Overall MAE

overall = 0
overall += pred_metric(prediction=train_preds,target=train.y, metrics='mae',␣
 ↪print_out=False, rescale_data=data)['mae']
overall += pred_metric(prediction=val_preds,target=val.y, metrics='mae',␣
 ↪print_out=False, rescale_data=data)['mae']
overall += pred_metric(prediction=test_preds,target=test.y, metrics='mae',␣
 ↪print_out=False, rescale_data=data)['mae']
print(f'Overall MAE: {overall/3}')
```

```
Overall MAE: 40.01786928641531
```

1.5.3 Residual Density

Furthermore, using the predictions from all three splits we can plot the residual density:

```
from grape_chem.plots import residual_density_plot
residual_density_plot(train_pred=train_preds, val_pred=val_preds,␣
 ↪test_pred=test_preds,
                      train_target=train.y, val_target=val.y, test_target=test.
 ↪y)
```



For the (scaled) residual density plot, we are looking to examine how our errors or residuals behave. Namely, we want them to be normally distributed, and they seem to be!

1.5.4 Latent analysis

PCA Finally, we could analyze the model latents by applying a PCA model and plotting them based on groups. Let us first classify a subset of our test SMILES and label them according to their classes:

```
from grape_chem.analysis import classyfire, classyfire_result_analysis
ids, data_ids = classyfire(test.smiles[:100], log=False)
# -> ids are used for the result analysis and data_ids for specifying the data␣
 ↪point from the test set we can use
```

```
class_dict, smile_dict, rel_ids = classyfire_result_analysis(idx=ids, layer=1,␣
↪return_relative_ids=True)
```

```
Found log file in working directory.
All passed smiles are already in the passed log_file.
Key error occurred using superclass for file 683.json.
Key error occurred using superclass for file 695.json.
Key error occurred using superclass for file 656.json.
Key error occurred using superclass for file 640.json.
Key error occurred using superclass for file 660.json.
Key error occurred using superclass for file 621.json.
Key error occurred using superclass for file 699.json.
Key error occurred using superclass for file 676.json.
```

```
indices = list(class_dict.keys())
labels = list(class_dict.values())
print(labels[:10])
```

```
['Benzenoids', 'Benzenoids', 'Benzenoids', 'Benzenoids', 'Phenylpropanoids and
polyketides', 'Benzenoids', 'Organic nitrogen compounds', 'Organophosphorus
compounds', 'Organoheterocyclic compounds', 'Organoheterocyclic compounds']
```

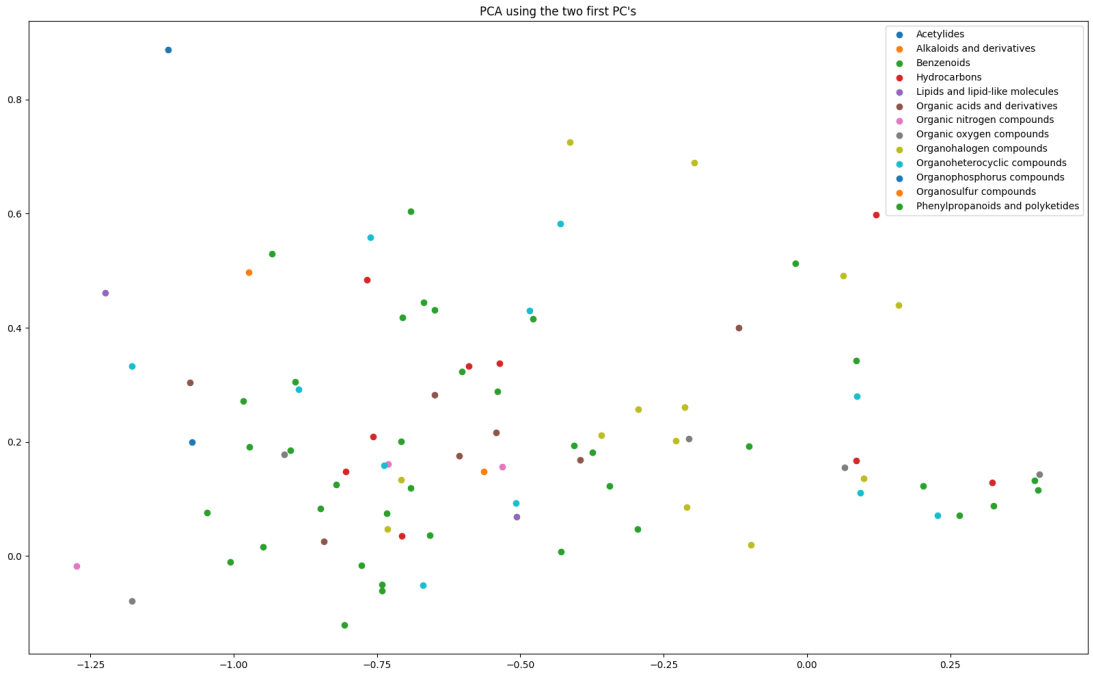
This will let us label the PCA plot much better. Now we can retrieve and generate the PCA plot:

```
lats_data = test[:100]
preds, lats = test_model(model, test_data_loader=lats_data, return_latents=True)
lats = lats.cpu().detach().numpy()
```

```
100%|      | 4/4 [00:00<00:00, 317.61it/s]
```

```
from grape_chem.plots import pca_2d_plot
pca_2d_plot(latents=lats[rel_ids], labels=labels, save_fig=True,␣
↪fig_size=(20,12))
```

```
<Axes: title={'center': "PCA using the two first PC's"}>
```

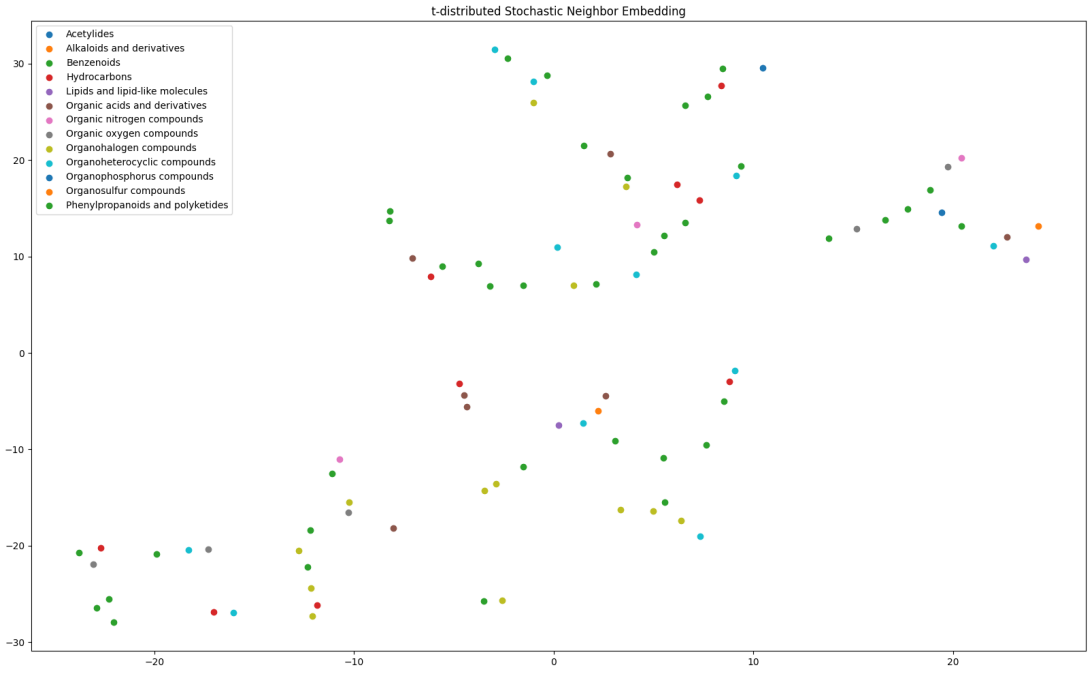


t-SNE Alternatively, we could use the t-SNE (t-distributed Stochastic Neighbor Embedding) algorithm by [4] to perform dimensionality reduction. The algorithm essentially uses probabilistic similarity measures to cluster the input point for n-components (here 2) and relies on two hyperparameters: perplexity (approximately equal to the number of neighbors per point) and the number of iterations the algorithm runs for.

It is generally a tricky feat to have the algorithm converge and produce interesting results, but it is possible! Here is an example use:

```
from grape_chem.plots import tSNE_plot
tSNE_plot(latents=lats[rel_ids], labels=labels, perplexity=5, save_fig=True,␣
↪fig_size=(20,12))
```

```
<Axes: title={'center': 't-distributed Stochastic Neighbor Embedding'}>
```



1.6 Prediction

The final step of the typical machine learning pipeline is to use the model for **prediction**. This is made easy by the `DataSet` method `predict_smiles`. The requirement for the method to work is that (1) the model and dataset share the same node- and edge-features (generally true if the model was trained using a specific `DataSet` object's data); and (2) that the input SMILES are valid compounds and recognized by `rdkit`.

Using it looks like the following:

```
SMILES_to_predict = ['CC','CCO', 'CCCCCC']
data.predict_smiles(SMILES_to_predict, model) # The results are in degrees␣
↪Celsius
```

```
{'CC': -244.16973876953125,
 'CCO': -88.8644027709961,
 'CCCCCC': -118.04553985595703}
```

Note that the results can only be rescaled if `DataSet` has internal `mean` and `std` value, or they have to be passed manually.

For completion, here is what it looks like to predict from a dataset *without* first training the model (using the FreeSolv dataset and AFP as an example):

```
from grape_chem.datasets import FreeSolv
from grape_chem.models import AFP
```

```
data = FreeSolv()
model = AFP(node_in_dim=data.num_node_features, edge_in_dim=data.
 ↪num_edge_features)
data.predict_smiles(['CC', 'CCCCCCCCC'], model)
```

```
{'CC': 0.0484485849738121, 'CCCCCCCCC': 0.05039355903863907}
```

These numbers are just random, of course.

1.7 References

[1] Jean-Claude Bradley and Andrew Lang and Antony Williams, Jean-Claude Bradley Double Plus Good (Highly Curated and Validated) Melting Point Dataset, 2014, http://dx.doi.org/10.6084/m9.figshare.1031637

[2] Feunang, Y., Eisner, R., Knox, C., Chepelev, L., Hastings, J., Owen, G., Fahy, E., Steinbeck, C., Subramanian, S., Bolton, E., Greiner, R., & Wishart, D. S. (2016). ClassyFire: automated chemical classification with a comprehensive, computable taxonomy. Journal of Cheminformatics, 8(1), 61. https://doi.org/10.1186/s13321-016-0174-y

[3] Butina, D. (1999). Unsupervised Data Base Clustering Based on Daylight's Fingerprint and Tanimoto Similarity: A Fast and Automated Way To Cluster Small and Large Data Sets. Journal of Chemical Information and Computer Sciences, 39(4), 747-750. https://doi.org/10.1021/ci9803381

[4] van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-sne. Journal of Machine Learning Research, 9 (86), 2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html

---

Extra Code to generate PDFs:

```
#PDF conversion code
!export PATH=/Library/TeX/texbin:$PATH
!jupyter nbconvert 'GraPE-Chem Demonstration'.ipynb --to pdf --no-prompt
```

```

```

F Advanced Demonstration of GraPE-Chem

Advanced GraPE-Chem Demonstration

June 10, 2024

1 Advanced Demo

The goal of this notebook is to demonstrate how `GraPE-Chem` for a more advanced workflow and in the context of hyperparameter optimization. Specifically, we will have a look at global features and demonstrating how we used the **BOHB** (Bayesian Optimization with HyperBand) by [1] as implemented in `Ray-Tune`.

1.1 Using Global Features

A global feature is a type of data that, in the context of molecules and graphs, describes extra, graph level information. Specifically, global features *are not* the target of whatever problem a GNN tries to solve. For instance, if we use the `QM9` dataset (link), we could use the *heat capacity* as our primary target and the *Dipole moment* as our global feature. These global features are then concatenated to the (last) latent representation of the GNN (after read-out). This simple introduction of the global features to the system has been shown to yield higher performance [2] without any new cost.

In the following, we will train a simple MPNN on the QM9 dataset using the heat capacity as our target and and the Dipole moment as a global feature.

```
from grape_chem.datasets import QM9

# The ids are based on the way it is stored in PyG
data = QM9(target_id=5, global_feature_id=0)
```

```
print('Some example features:\n')
print('SMILES: ', data.smiles[:10], '\n')
print('Heat capacity: ', data.target[:10], '\n')
print('Dipole moment: ',data.global_features[:10], '\n')
```

```
Some example features:

SMILES:  ['C#C' 'C#N' 'C=O' 'CC' 'CO' 'C#CC' 'CC#N' 'CC=O' 'NC=O' 'CCC']

Heat capacity:  [ 59.5248  48.7476  59.9891 109.5031  83.794  177.1963 160.7223
166.9728
 145.3078 227.1361]

Dipole moment:  tensor([0.0000, 1.6256, 1.8511, 0.0000, 2.8937, 2.1089, 0.0000,
```

```
1.5258, 0.7156,
        3.8266])
```

To reduce the computation time for this example, we will only consider a subset of 3000 SMILES from QM9. However, one can just skip the following step if the whole dataset is to be used:

```
from grape_chem.utils import DataSet


data = DataSet(smiles=data.smiles[:3000], target=data.target[:3000],
                            global_features=data.global_features[:3000])
```

We now prepare the data as usual:

```
train, val, test = data.split_and_scale(scale=True, split_type='random',␣
 ↪seed=1234)
```

Initializing the model:

```
from grape_chem.models import MPNN

model = MPNN(node_in_dim=data.num_node_features, edge_in_dim=data.
 ↪num_edge_features, num_global_feats=1)
```

Note that we must now specify how many global features we are adding, in addition to how many node and edge features we have. But the rest stays the same.

Let us now specify all the required training elements:

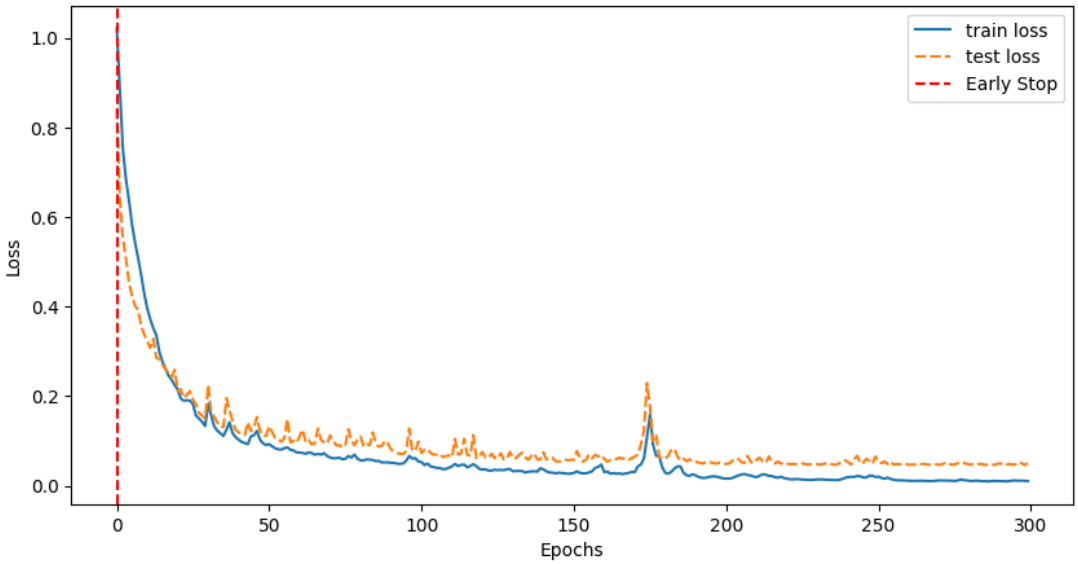
```
import torch
from grape_chem.utils import EarlyStopping
from torch.optim import lr_scheduler

# optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-6)


# Early stopper
early_stopper = EarlyStopping(patience=30, model_name='best_model')

# Learning rate scheduler
scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.9,␣
 ↪min_lr=0.0000000000001, patience=15)
```

We now train and plot the training/validation loss:

```
from grape_chem.utils import train_model

train_loss, val_loss = train_model(model = model,
```

```
                                    loss_func = 'mse',
                                    optimizer = optimizer,
                                    train_data_loader= train,
                                    val_data_loader = val,
                                    batch_size=300,
                                    epochs=300,
                                    early_stopper=early_stopper,
                                     scheduler=scheduler)

model.load_state_dict(torch.load('best_model.pt'))

from grape_chem.plots import loss_plot
loss_plot([train_loss, val_loss], ['train loss', 'test loss'], early_stopper.
 ↪stop_epoch)
```

```
epoch=298, training loss= 0.011, validation loss= 0.047: 100%|      |
300/300 [02:47<00:00,  1.79it/s]
```



As one might be able to tell from the above code, the only two things that changed was (1) we need to initialize/use data that has a valid global feature and (2) we need to specify how many global features we are adding to the model.

If global features are available, it is worth adding them!

1.2 BOHB and Hyperparameter Optimization

In this second part of the advanced demonstration, we will have a look at hyperparameter optimization, how GraPE-Chem fits in and Ray-Tune. Note that for this, we need to install the packages:

`ray`, `ConfigSpace==0.4.18` and `hpbandster==0.7.4`.

Let us briefly review what the core of the BOHB algorithm is. Essentially, it consists of two parts: Bayesian Optimization [3] (BO-) and HyperBand [4] (-HB), each acting as the search algorithm and a trial scheduler respectively. Bayesian optimization leverages Gaussian processes to sample the search space in a 'smart' way that optimizes the evaluation metric, and while it is very powerful, it can take a long time. To offset this, BOHB adds HyperBand, which is a bandit-based trial scheduler, capable of terminating and restarting trials to maximize trial run-time efficiency. Together, these two algorithm profit from each others strengths.

For more information on the algorithm itself please see their GitHub repo (github) or their blog post reviewing their paper (blog-post).

1.2.1 Ray Tune

Ray (https://github.com/ray-project/ray) and specifically Ray Tune is a package offers streamlined hyperparameter optimization. For hyperparameter optimization using it, we need three things: * a parameter search space * an objective function which contains training and validation * a ray tuner

For more information on how to use BOHB with ray-tune please check out (using BOHB) and (using checkpoints).

The model we will be using is AFP and the dataset is FreeSolv. As for hyperparameters, let us optimize the learning rate, the hidden (node) representation size and the number of hidden output (MLP) layers.

Let us start with defining the search space:

```
import ConfigSpace as CS
config_space = CS.ConfigurationSpace()
config_space.add_hyperparameter(CS.UniformIntegerHyperparameter("mlp_out",␣
 ↪lower=1, upper=5))
config_space.add_hyperparameter(CS.
 ↪UniformIntegerHyperparameter("gnn_hidden_dim", lower=32, upper=256))
config_space.add_hyperparameter(CS.UniformFloatHyperparameter('initial_lr',␣
 ↪lower=1e-5, upper=1e-1));
```

Now we need to define an objective function. Specifically, this function needs to (1) take a configuration dictionary `config` holding the trial hyperparameters, (2) load a model based on that config-dictionary and (3) load the correct data. All of this needs to happen **inside** the function, as Tune will need to load multiple different instances of data and models.

```
import os
import tempfile
from grape_chem.models import AFP
from grape_chem.datasets import FreeSolv
from grape_chem.utils import return_hidden_layers, train_epoch, val_epoch,␣
 ↪set_seed
from torch_geometric.loader import DataLoader
```

```
from ray.train import Checkpoint

# Seed everything
set_seed(42)

def trainable(config: dict, device:torch.device):
        """ The trainable for Ray-Tune.

        Parameters
        ----------
            config: dict
                A ConfigSpace dictionary adhering to the required parameters in␣
 ↪the trainable. Defines the search space of the HO.
            device: torch.device
        """

        ################ load the data ###############
        data = FreeSolv()
        train_set, val_set, _ = data.split_and_scale(scale=True,␣
 ↪split_type='random')
        train_data = DataLoader(train_set, batch_size = 300)
        val_data = DataLoader(val_set, batch_size = 300)

        ################ load the model ###############

        # Get mlp hidden layers
        mlp = return_hidden_layers(config['mlp_out'])

        # Define the model
        model = AFP(node_in_dim=data.num_node_features,edge_in_dim=data.
 ↪num_edge_features, hidden_dim=config['gnn_hidden_dim'], mlp_out_hidden=mlp)
        model.to(device=device)


        # We define training stuff
        optimizer = torch.optim.Adam(model.parameters(),␣
 ↪lr=config['initial_lr'], weight_decay=1e-6)
        early_Stopper = EarlyStopping(patience=30, model_name='random',␣
 ↪skip_save=True)
        scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',␣
 ↪factor=0.999, min_lr=1e-10, patience=10)
        loss_function = torch.nn.functional.l1_loss

        iterations = 300

        start_epoch = 0
```

```
        # HyperBand uses checkpoint, so we need to check whether to load one
        checkpoint = train.get_checkpoint()

        if checkpoint:
            with checkpoint.as_directory() as checkpoint_dir:
                model_state_dict = torch.load(
                    os.path.join(checkpoint_dir, "model.pt"),
                    # map_location=...,  # Load onto a different device if␣
↪needed.
                )
                model.module.load_state_dict(model_state_dict)
                optimizer.load_state_dict(
                    torch.load(os.path.join(checkpoint_dir, "optimizer.pt"))
                )
                start_epoch = (
                    torch.load(os.path.join(checkpoint_dir, "extra_state.
↪pt"))["epoch"] + 1
                )

        model.train()

        for i in range(start_epoch, iterations):
            # We take a training step
            train_loss = train_epoch(model=model, loss_func=loss_function,␣
↪optimizer=optimizer,train_loader=train_data, device=device)
            # Then we validate that step
            val_loss = val_epoch(model=model, loss_func=loss_function,␣
↪val_loader=val_data, device=device)
            scheduler.step(val_loss)

            # We check if we reached early stopping, and if so break out of the␣
↪loop
            early_Stopper(val_loss=val_loss, model=model)
            if early_Stopper.stop:
                train.report({"mae_loss": val_loss})
                break

            with tempfile.TemporaryDirectory() as temp_checkpoint_dir:
                checkpoint = None

                # We save a checkpoint if we don't early stop and are on n*15␣
↪epoch
                should_checkpoint = i % config.get("checkpoint_freq", 15) == 0
                # In standard DDP training, where the model is the same across␣
↪all ranks,
```

```
                # only the global rank 0 worker needs to save and report the␣
↪checkpoint
                if train.get_context().get_world_rank() == 0 and␣
↪should_checkpoint:
                    # === Make sure to save all state needed for resuming␣
↪training ===
                    torch.save(
                        model.module.state_dict(),  # NOTE: Unwrap the model.
                        os.path.join(temp_checkpoint_dir, "model.pt"),
                    )
                    torch.save(
                        optimizer.state_dict(),
                        os.path.join(temp_checkpoint_dir, "optimizer.pt"),
                    )
                    torch.save(
                        {"epoch": i},
                        os.path.join(temp_checkpoint_dir, "extra_state.pt"),
                    )
                    #␣
↪==================================================================
                    checkpoint = Checkpoint.from_directory(temp_checkpoint_dir)

                # We report the training criteria
                train.report({"mae_loss": val_loss}, checkpoint=checkpoint)
```

```
[W ParallelNative.cpp:230] Warning: Cannot set number of intraop threads after
parallel work has started or after set_num_threads call when using native
parallel backend (function set_num_threads)
```

The warning above is because we are working in the jupyter environment. The final building block to optimize is the `Tuner`, essentially the object that controls the training procedure:

```
from functools import partial
from ray import tune, train
from ray.tune.search.bohb import TuneBOHB
from ray.tune.schedulers.hb_bohb import HyperBandForBOHB

if torch.cuda.is_available():
    device = torch.device("cuda")
    gpu = 1
else:
    device = torch.device("cpu")
    gpu = 0

# Partially initialize the trainable
my_trainable = partial(trainable, device=device)

# Give the tuner CPU and GPU resources
```

```
trainable_with_resources = tune.with_resources(my_trainable, {"cpu":1, "gpu":
 ↪gpu})

### Define search algorithm to be BOHB
algo = TuneBOHB(config_space,mode='min', metric="mae_loss",)

## Get the trial control algorithm
scheduler = HyperBandForBOHB(
    time_attr="training_iteration",
    max_t=10, # The iterations allowed per instance
    )

n_samples = 5 # These are the number of trials, ie. we should increase this for␣
 ↪better results.

## Initialize the tuner
tuner = tune.Tuner(trainable_with_resources,
              tune_config=tune.TuneConfig(
                                    scheduler=scheduler,
                                    search_alg=algo,
                                    mode='min',
                                    metric="mae_loss",
                                    num_samples=n_samples),
              run_config=train.RunConfig(
                                    name="bo_exp",
                                    stop={"training_iteration": 100}),
)
```

That's it! The above code is (mostly) dataset and model agnostic, so we can freely change them around. Furthermore, we can either run this locally or remote (HPC) and scale it up as much as we want. But we have to be a bit careful if we want to switch out the algorithm or scheduler, as this will change some small details.

Let us run the optimizer:

```
result = tuner.fit(); # We suppress the output because it is quite long, but␣
 ↪feel free to have a look yourself.
```

```
<IPython.core.display.HTML object>
```

```
(func pid=16485) Downloading https://deepchemdata.s3-us-
west-1.amazonaws.com/datasets/SAMPL.csv
(func pid=16485) Processing…
(func pid=16485) Done!
(func pid=16490) Downloading https://deepchemdata.s3-us-
```

```
west-1.amazonaws.com/datasets/SAMPL.csv [repeated 2x across cluster] (Ray
deduplicates logs by default. Set RAY_DEDUP_LOGS=0 to disable log deduplication,
or see https://docs.ray.io/en/master/ray-observability/user-guides/configure-
logging.html#log-deduplication for more options.)
(func pid=16490) Processing… [repeated 2x across cluster]
(func pid=16490) Done! [repeated 2x across cluster]
(func pid=16503) Downloading https://deepchemdata.s3-us-
west-1.amazonaws.com/datasets/SAMPL.csv [repeated 2x across cluster]
(func pid=16503) Processing… [repeated 2x across cluster]
(func pid=16492) Done!
(func pid=16503) Done!
2024-06-04 16:16:06,231 INFO hyperband.py:543 -- Restoring from a previous point
in time. Previous=1; Now=1
2024-06-04 16:16:09,085 INFO hyperband.py:543 -- Restoring from a previous point
in time. Previous=1; Now=1
2024-06-04 16:16:11,995 INFO hyperband.py:543 -- Restoring from a previous point
in time. Previous=1; Now=1
2024-06-04 16:16:16,912 INFO hyperband.py:543 -- Restoring from a previous point
in time. Previous=3; Now=1
2024-06-04 16:16:18,292 WARNING experiment_state.py:205 -- Experiment state
snapshotting has been triggered multiple times in the last 5.0 seconds. A
snapshot is forced if `CheckpointConfig(num_to_keep)` is set, and a trial has
checkpointed >= `num_to_keep` times since the last snapshot.
You may want to consider increasing the `CheckpointConfig(num_to_keep)` or
decreasing the frequency of saving checkpoints.
You can suppress this error by setting the environment variable
TUNE_WARN_EXCESSIVE_EXPERIMENT_CHECKPOINT_SYNC_THRESHOLD_S to a smaller value
than the current threshold (5.0).
2024-06-04 16:16:18,294 INFO tune.py:1007 -- Wrote the latest version of all
result files and experiment state to '/Users/faerte/ray_results/bo_exp' in
0.0050s.
2024-06-04 16:16:18,299 INFO tune.py:1039 -- Total run time: 31.83 seconds
(31.80 seconds for the tuning loop).
```

```python
print('Best config is: ', result.get_best_result().config)
```

```
Best config is:  {'gnn_hidden_dim': 120, 'initial_lr': 0.007187876555629309,
'mlp_out': 2}
```

And that is our best set of hyperparameters for this (small) set of trials! Like mentioned, we expect the result to get better and better (up to a certain point) as we increase the number of trials.

1.3 References:

[1] Falkner, S., Klein, A., & Hutter, F. (2018). Bohb: Robust and efficient hyperparameter optimization at scale.

[2] Na, G. S., Kim, H. W., & Chang, H. (2020). Costless performance improvement in machine learning for graph-based molecular analysis [PMID: 31928003]. Journal of Chemical Information and Modeling, 60 (3), 1137–1145. https://doi.org/10.1021/acs.jcim.9b00816

[3] Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & de Freitas, N. (2016). Taking the human out of the loop: A review of bayesian optimization. Proceedings of the IEEE, 104 (1), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

[4] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2018). Hyperband: A novel bandit-based approach to hyperparameter optimization.

Extra Code to generate PDFs:

```
#PDF conversion code
!export PATH=/Library/TeX/texbin:$PATH
!jupyter nbconvert 'Advanced GraPE-Chem Demonstration'.ipynb --to pdf␣
 ↪--no-prompt
```

```
[NbConvertApp] Converting notebook Advanced GraPE-Chem Demonstration.ipynb to
pdf
[NbConvertApp] Support files will be in Advanced GraPE-Chem Demonstration_files/
[NbConvertApp] Making directory ./Advanced GraPE-Chem Demonstration_files
[NbConvertApp] Writing 64745 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 121333 bytes to Advanced GraPE-Chem Demonstration.pdf
```